Supporting Automated Containment Checking of Software Behavioural Models Using Model Transformations and Model Checking

Faiz UL Muram[a,b,*], Huy Tran[a], Uwe Zdun[a]

[a] *University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria*
[b] *School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden*

Abstract

Models are extensively used in many areas of software engineering to represent the behaviour of software systems at different levels of abstraction. Because of the involvement of different stakeholders in constructing these models and their independent evolution, inconsistencies might occur between the models. It is thus crucial to detect these inconsistencies at early phases of the software development process, and especially as soon as refined models deviate from their abstract counterparts. In this article, we introduce a containment checking approach to verify whether a certain low-level behaviour model, typically created by refining and enhancing a high-level model, still is consistent with the specification provided in its high-level counterpart. We interpret the containment checking problem as a model checking problem, which has not received special treatment in the literature so far. Because the containment checking is based on model checking, it requires both formal consistency constraints and specifications of these models. Unfortunately, creating formal consistency constraints and specifications is currently done manually, and therefore, labour-intensive and error prone. To alleviate this issue, we define and develop a fully automated transformation of behaviour models into formal specifications and properties. The generated formal specifications and properties can directly be used by existing model checkers for detecting any discrepancy between the input models and yield corresponding counterexamples. Moreover, our approach can provide the developers more informative and comprehensive feedback regarding the inconsistency issues, and therefore, help them to efficiently identify and resolve the problems. The evaluation of various scenarios from industrial case studies demonstrates that the proposed approach efficiently translates the behaviour models into formal specifications and properties.

Keywords: Containment Checking, Consistency Checking, Model Checking, Model Transformation, Behaviour Models, LTL

1. Introduction

During the modelling phase of a software system, behaviour models such as UML activity diagrams, sequence diagrams, statecharts [1], Business Process Model and Notation (BPMN) [2], and Event Driven Process Chain (EPC) [3] 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 [4]. 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 and 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.

[*]Corresponding author

Email addresses: `faizul.muran@univie.ac.at, faiz.ul.muram@mdh.se` (Faiz UL Muram), `huy.tran@univie.ac.at` (Huy Tran), `uwe.zdun@univie.ac.at` (Uwe Zdun)

This has led to a rich body of work for checking and managing model consistency in the literature [4–6]. Of these existing approaches, only a few deal with consistency checking of behavioural models for software systems [4, 5], for instance, checking behavioural models against non-behavioural models [7–9], or checking different types of behaviour models [10–13]. 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 [14–16]. 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. [4]) is to describe the behaviour models under study in form of formal specifications and derive consistency constraints that these specifications must satisfy. There are two key shortcomings that hinder the wider applicability of the current state-of-the-art consistency checking approaches. Firstly, 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 [11]. Moreover, this task is often accomplished in a laborious and manual manner, which is also error-prone [13, 17]. Secondly, the results produced by existing model checkers (e.g., counterexamples) are rather cryptic and verbose. As a consequence, they are difficult for the developers, who—as mentioned above—often have limited knowledge of the underlying formal techniques, to interpret and understand [18, 19]. It is also necessary to investigate the applicability and technical feasibility of the approach for realistically sized of models.

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

- **RQ1**: How to perform automated transformation of behaviour models into formal specifications and descriptions?
- **RQ2**: How can we effectively communicate the containment checking results to the developers?
- **RQ3**: 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 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 [20–23], no existing techniques can be leveraged for accomplishing Task **T1**. We propose an automated method for transforming the high-level and low-level behaviour models into temporal logic based consistency constraints and formal behaviour descriptions, respectively. In particular, Linear Temporal Logic (LTL) [24] and the state based SMV (Symbolic Model Verifier) language are used for formalising the input models, and the NuSMV (New Symbolic Model Verifier) model checker [25] is used for verification. The behaviour models studied in our work are Unified Modeling Language (UML) 2.4.1 activity diagrams [1] as they are widely used in both academia and industry for modelling and analysing behaviours of software systems.

To address the second research question, our third Task **T3** concentrates on extracting more meaningful information from the containment checking results produced by model checking in order to facilitate better understanding and resolving of the violations revealed by containment checking. 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. In particular, 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 order to address the third research question, we have applied our approach in scenarios of different sizes and complexity from four industrial case studies representing real systems from the banking and e-business domains. The scenarios with few dozens to hundreds of elements including multiple interleaving and iterative activities are typical sizes of models used in industry by developers [26]. 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.

In our earlier works, the initial ideas were discussed for addressing the containment checking problem for behaviour models using model checking techniques [27, 28]. The contributions presented in this article significantly extend the prior works in several aspects. First, we devise a more 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 as in [27]. Second, this work covers a more comprehensive set of modelling constructs used for describing software behaviour. Apart from the fundamental set of modelling constructs considered in [27, 28], we investigate complex structures 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 present in detail the counterexample analysis method for locating the cause(s) of inconsistency and presenting the appropriate suggestions and guidelines to aid developers in resolving containment inconsistencies. Fourth, 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 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 article is structured as follows. Section 2 discusses the state-of-the-art regarding behavioural consistency checking techniques in general and containment checking in particular, and formal semantics of behavioural models. Section 3 discusses the background information. In Section 4, we describe our approach for containment checking based on model checking in detail. The scenario extracted from industrial case study is described in depth to illustrate our approach in Section 5. In Section 6, 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 7, we discuss various aspects and challenges of supporting containment checking. Finally, Section 8 concludes with some final remarks and future research directions.

2. State-of-the-Art

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.

2.1. Existing Approaches for Consistency Checking

In the literature, many approaches tackled different types of models and/or model checking techniques [4–6]. 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) [9]. 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 [29]. 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 [30]. 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 [11, 12]. 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 SPIN (Simple Promela Interpreter) model checker [11]. 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 UPPAAL model checker [12]. Amálio et al. [31] 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/Eves theorem prover.

Lam and Paget propose an algebraic approach for verifying the consistency between sequence diagrams and statecharts. In this approach, behaviour models are encoded into π-calculus to support the verification of model consistency [13]. Wang et al. introduce an approach for consistency checking between UML 1.5 statecharts and sequence diagrams [10]. 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 [32]. The $RSML^{-e}$ (Requirements State Machine Language) is used for the specification of the systems which is then translated into NuSMV 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" [6, 33]. 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 [34]. 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) [35, 36]. 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 [37] 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 [38]. Bae et al. propose quantitative process similarity metric for measuring mining process similarity and difference [39]. 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. 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.

2.2. *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 [17]. Van der Aalst presents a theoretical framework for defining the semantics of behaviour inheritance [40]. 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. [41] 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 FDR 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 [42]. This approach alone is not applicable for containment checking, as behaviour models contain control constructs that cannot be matched only structurally. Arcaini et al. [43] and Krings and Leuschel [44] 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 Yices is used to prove refinement correctness. The later focusses on the implementation of BMC (Bounded Model Checking), k-Induction and IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness) 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 ProB. However, these approaches consider formal specification languages (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, as well as the counterexample analysis for locating the cause(s) of containment problems and their resolutions. The idea with this research is to solve this 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.

2.3. *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 [45]. 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 [46]. 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 [47]. To perform verification, Business Process Execution Language (BPEL) [48] 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 [7]. 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. In our approach, 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 PPSL (Process Pattern Specification Language) [49]. 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 (Dynamic Meta Modelling) rules and GROOVE (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 [50]. Wasylkowski and Zeller propose the concept of operational preconditions along with a tool named Tikanga [51]. 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. LTL model checkers can be classified as explicit or symbolic. Explicit model checkers such as SPIN[1] 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 [52]. 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 [25] model checker used in this work is based on the symbolic model checking technique, and therefore, is able to support the verification of large systems up to 10^{20} states [53]. 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) and analysis of the counterexample for locating the causes of inconsistencies, thus, does not require the strong knowledge of formal methods.

2.4. *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 [54]. 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 [55]. Eshuis and Wieringa present the formal semantics of UML 1.3 activity diagrams for workflow models based on STATEMATE semantics (see [56]) of statecharts and also introduce data integrity constraints [8]. In [20], 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 [21], 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

[1]See http://spinroot.com

Table 1: Overview and Comparison of Related Work

Reference	**Consistency Type**	**Model Type**	**UML Version**	**Translation to Formal Descriptions**	**Automation of Consistency Checking**
Yeung [9]	horizontal	statechart, class diagram	UML 1.4	manual	-
Van der Straeten et al. [29]	evolution-horizontal	statechart, class diagram, sequence diagram	UML 1.5	manual	automated
Graaf and Van Deursen [30]	horizontal	statechart, sequence diagram	UML 1.4	semi-automated	manual
Amálio et al. [31]	horizontal	statechart, class diagram, snapshot diagram	–	manual	automated
Knapp et al. [11, 12]	horizontal	statechart, collaboration diagram	UML 1.4	semi-automated	automated
Lam et al. [13]	horizontal	statechart, sequence diagram	UML 2.0	semi-automated	automated
Wang et al. [10]	horizontal	statechart, sequence diagram	UML 1.5	semi-automated	automated
Heimdahl et al. [32]	horizontal	$RSML^{-e}$	–	semi-automated	automated
Dijkman et al. [35, 36]	horizontal	process model	–	semi-automated	automated
Bae et al. [39]	horizontal	process model	–	semi-automated	automated
Stumptner and Schrefi [17]	vertical	statechart	UML 1.3	semi-automated	automated
Van der Aalst [40]	vertical	statechart, activity diagram, sequence diagram	UML 1.4	semi-automated	automated
Egyed [42]	vertical-structure containment	class diagram	UML 1.3	automated	automated
Arcaini et al. [43]	vertical	abstract state machine	-	automated	automated
Krings and Leuschel [44]	vertical	-	-	-	automated
Van der Aalst et al. [37]	horizontal	process model	–	manual	automated
Van der Aalst et al. [38]	horizontal	process model	–	semi-automated	automated
Engels et al. [46]	horizontal	activity diagram, visual contracts	UML 2.1	semi-automated	automated
Engels et al. [41]	horizontal-vertical	capsule and protocol statecharts	UML 1.4	manual	automated
Koehler et al. [45]	vertical	process model	–	semi-automated	automated
Förster et al. [49]	horizontal	process model	UML 2.0	automated	automated
Wasylkowski and Zeller [51]	horizontal	Java program	–	semi-automated	automated
Janssen et al. [50]	horizontal	process model	–	automated	automated
Martens [47]	vertical	process model	–	semi-automated	automated
Tsiolakis and Ehrig [7]	horizontal	class diagram, sequence diagram	UML 1.3	manual	automated
Yang and Shensheng [54]	horizontal	activity diagram	UML 1.4	manual	automated
Eshuis and Wieringa [8]	–	class diagram, activity diagram	UML 1.3	–	–
Eshuis [21]	horizontal	class diagram, activity diagram	UML 1.5	semi-automated	automated
Eshuis and Wieringa [20]	horizontal	activity diagram	UML 1.4	semi-automated	automated
Guelfi and Mammar [55]	horizontal	activity diagram	UML 2.0	manual	automated
Lam [23]	horizontal	activity diagram	UML 2.1	manual	–
Lam [22]	horizontal	activity diagram	UML 2.1	manual	automated
Our Approach	vertical-behaviour containment	activity diagram	UML 2.4.1	automated	automated

nodes. These approaches tackle UML 1.x semantics based on statecharts; although activity diagrams and statecharts seem syntactically similar, every activity diagram cannot be transformed into a statechart [55]. Besides that, UML 2 activity diagrams have different semantics based on Petri nets [55, 57]. 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 [55].

Lam introduces theoretical foundations for specifying and classifying different equivalence notions of a subset of UML activity diagrams [23]. In [22] 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 [20], we abstract and encode guards and 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 [58], 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 1 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, CTL, and TCTL subset etc.) and counterexample analysis are not presented in this table. The minus ($-$) symbols in the table represents the unavailability of information regarding a particular activity. The table demonstrates that none of the published approaches have considered the containment checking problem for behaviour models. Most of the existing approaches use UML version 1.x instead of UML 2. It might be noted that automatic translation to formal consistency constraints and counterexample analysis have also not been considered in existing approaches. Accordingly, this research focuses on automatic translation of behaviour models into consistency constraints and formal descriptions, containment checking and counterexample analysis.

3. Preliminaries

3.1. Linear Temporal Logic

In our study we choose Linear Temporal Logic (LTL) [24] for specifying the temporal relationships between the involved elements of the high-level activity diagram. 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 UML models because they enable explicit reasoning about states and transition executions of the input models for containment relationships. In the mapping scheme of constructs of UML activity diagrams into LTL, 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).

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

In the context of containment checking, we consider the appearance of each basic action of a UML activity diagram in the execution path under study as a primitive proposition. The syntax and standard semantics for LTL operators are as follows: The negation ($\neg\varphi$) states that formula φ holds if and only if it does not hold in the first state of the path. The conjunction in LTL formula $\varphi_1 \wedge \varphi_2$ holds if and only if both φ_1 and φ_2 are true; however, the disjunction $\varphi_1 \vee \varphi_2$ according to the DeMorgan's rules is equivalent to $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$. $\mathbf{X}\varphi$ ("ne**X**t") states that φ will hold in the next state. $\mathbf{F}\varphi$ ("**F**uture/Finally") means that formula φ will hold in some future state. $\mathbf{G}\varphi$ ("**G**lobally/Always") indicates that formula φ will continuously hold in all future states. The formula φ_1 **U** φ_2 ("**U**ntil") 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. φ_1 **R** φ_2 ("**R**elease") states that φ_2 is true until φ_1 becomes true, or φ_2 is true forever. The operator **Y**φ ("**Y**esterday/Previous") means that formula φ held in the previous state. **O**φ ("**O**nce") states that formula φ has happened at sometime in the past. **H**φ ("**H**istorically") indicates that formula φ always held in the past. The formula φ_1 **S** φ_2 ("**S**nice") holds if φ_2 held at some moment in the past and, since then, φ_1 held all the time. It might be noted that, φ_2 must have been true at some point in the past in order to hold φ_1 **S** φ_2.

We adopt the conventional semantics of LTL as defined in the field of model checking [59]. 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 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 [60]. The size explosion in removing the past operators affects the complexity of containment because the size of the formula is part of the complexity. However, it 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 "$(\neg b \text{ U } (a \wedge \neg b))$".

3.2. NuSMV Overview

The NuSMV model checker supports symbolic model verification [61] and is widely used in both academia and industry [52]. The language that underpins the formal descriptions is hereafter called SMV specification language, which allows the description of Finite State Machines (FSMs). The basic purpose of the SMV description language is to describe the transition relation of a finite Kripke structure. The SMV description language 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. In particular, each instance of a module is processed synchronously by default with the others during an execution. However, NuSMV can also support interleaving concurrency in module instantiation. To get different instances of a module, instantiations can be parameterized. The variables can be of type boolean, enumerative or integers. For example, a variable c is declared as a boolean (`c : boolean;`), which can take either the value 0 or 1; the variable e is a scalar (`e: e1,e2,e3;`), which can take one of the symbolic values `e1, e2, or e3`. The variable *cal* is declared as a signal (`cal : 0..5;`), which can take any value inclusively in the range from 0 to 5. SMV also supports array as a data type (`ar : array 2..0 of boolean;`). The predicates use the logical operators *AND* ("&"), *OR* ("|") and *NOT* ("!"). The complete syntax and semantics definitions of SMV description can be found in [62].

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 initialised with a `false` value (except `Initial Nodes`). It can move to a different state (e.g. `true`) if the incoming conditions are satisfied. 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`. An assignment can be a non-deterministic choice by putting the multiple expressions in curly brackets ({}). Another way of defining the transition relation is to use a predicate that can refer to the current and next values of the variables. This transition predicate is preceded by keyword `TRANS`.

3.3. Activity Diagram

The definition of an activity diagram is based on the definition of classical transition systems [63]. An activity diagram needs to be adequately accommodate relevant concepts of a UML activity diagram such as different kinds of nodes, edges, and guards. Actions are the basic elements of an activity diagram and they are always atomic (i.e., they cannot be broken down further within the activity).

Definition 1 (Activity model)**.** *An activity model $\mathscr{A}$ is a tuple (N, E, G) where*

- *N is a finite set of nodes, derived from the UML 2 specification [1, Sec. 12]*
- *$E \subseteq N \times N$ is an ordered finite set of edges,*
- *G is a finite set of guard expressions,*
- *$N \rightarrow$ {*InitialNode, ActivityFinalNode, FlowFinalNode, Action, DecisionNode, MergeNode, ForkNode, JoinNode, LoopNode, ConditionalNode, ParameterNode, Interruptible Activity Region*}.*
- *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 $s,t \in N$.*

4. 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.

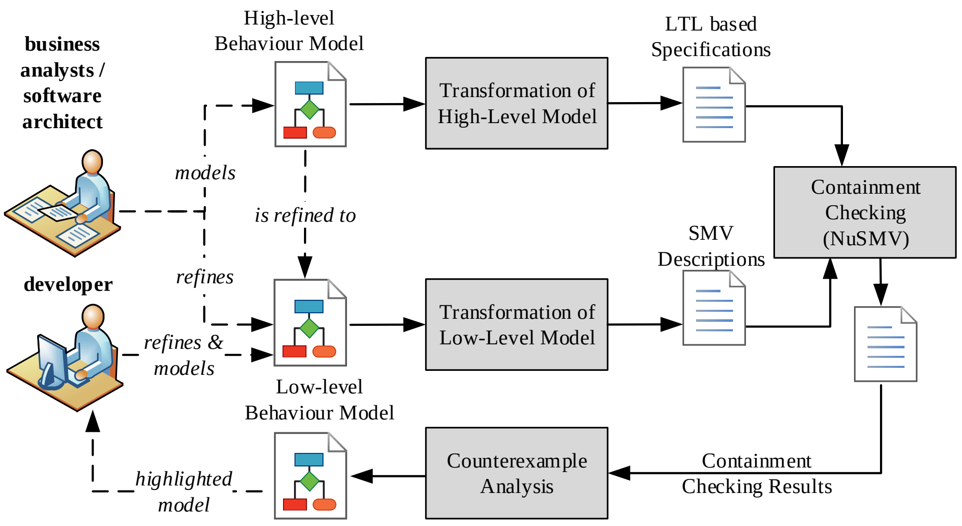

business analysts / software architect
High-level Behaviour Model
Transformation of High-Level Model
LTL based Specifications
models
is refined to
Containment Checking (NuSMV)
SMV Descriptions
developer
refines
refines & models
Transformation of Low-Level Model
Low-level Behaviour Model
highlighted model
Counterexample Analysis
Containment Checking Results

Figure 1: Overview of the Containment Checking Approach

An overview of our approach is shown in Figure 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 [61]. 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. Next, the generated counterexamples are scrutinised to uncover the causes of containment inconsistencies and provide feedback to help the developers to resolve the issues. In the subsequent sections, we explain these steps in detail. The behaviour models considered in our study are UML activity diagrams [1]. 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.

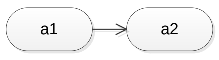
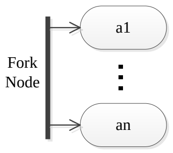
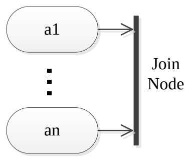
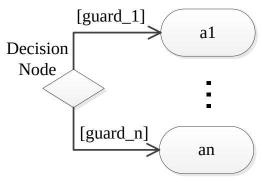
4.1. Step 1: Generating Consistency Constraints from the High-Level Model

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 [1], 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 [64], 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 a 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 an additional condition using decision and merge (conditional insert).

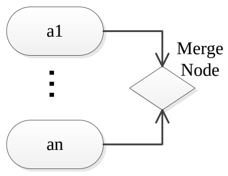
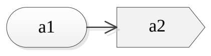
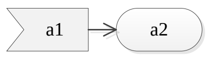
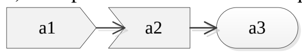
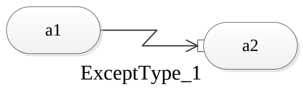
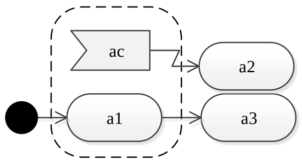
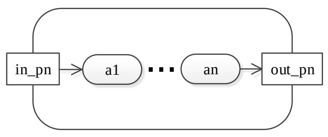
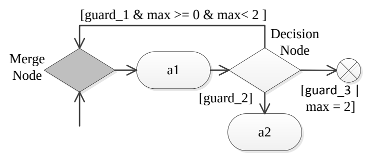
The creation of LTL formulas for containment or consistency checking is a very knowledge intensive endeavour. Therefore, we perform the automated creation of the formal constraints by defining LTL-based rules for formally representing constructs of activity diagrams based on containment relationship. Then our approach can take an input UML activity diagram and automatically transform it into corresponding LTL formulas using the LTL-based transformation rules. Table 2 summarises the constructs of UML activity diagrams along with their informal descriptions extracted from the UML 2 specification [1].

Table 2: Transformation Rules for Generating LTL Formulas for Containment of UML Activity Diagrams

UML Constructs	**Modelling Notation**	**LTL-Based Transformation Rules**
Sequencing of Actions: A set of actions (transitively) executed in sequential order. For instance, the execution of $a1$ will trigger the execution of $a2$.		`G (a1 -> F a2)`
Fork Node: The execution of a Fork Node leads to the parallel execution of subsequent actions ($a1...an$) [1, p. 387]. The semantics of Fork Node describes that all actions are executed simultaneously without any restriction.		`G (ForkNode -> F (a1 & ... & an)) & G ((a1 & ... & an)-> O ForkNode )`
Join Node: The concurrent execution of multiple actions ($a1...an$) are followed by the execution of a Join Node [1, p. 393]. Specifically, a Join Node is used to synchronize incoming concurrent flows.		`G ((a1 & ... & an)-> 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 [1, p. 370]. It is assumed that only one of the guards of the outgoing control flows evaluates to true at a time.		`G (DecisionNode -> F (a1 xor ... xor an))` or equivalently, but more complex `(DecisionNode -> F ((a1 & ! ... & ! an)| (! a1 & !... & an) ))`

(continued on next page)

Table 2: Transformation Rules for Generating LTL Formulas for Containment of UML Activity Diagrams

UML Constructs	**Modelling Notation**	**LTL-Based Transformation Rules**
Merge Node: The execution of exclusively one action among a set of alternative actions will lead to the execution of a Merge Node [1, p. 398].		`G (a1 xor ... xor an -> F MergeNode)` or equivalently, but more complex `G (((a1 & ! ... & !an)| ... | (!a1 & !... & an))-> F MergeNode)`
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. [1, p. 421]. In particular, the semantics shows that Send Signal Action instantly read the event from its inputs.		`G (a1 -> X a2)`
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 [1, p. 317]. 2) Accept Event Action with incoming flows waits to receive an input and enables only after the signal is sent by the prior action [1, p. 317]. Afterwards, Accept Event Action leads to the execution of next action.	 1) Accept Event Action with no input  2) Accept Event Action with incoming flows	1)`G (a1 -> X a2)& G !(a1 & a2)` 2)`G (a1 -> G (a2 -> X a3))`
Exception Handler: Exception Handler leads to execution of the handler body in case the specified exception occurs during the execution of the protected node [1, p. 373]. A variable, namely, `ExceptionType_i` is specified for handling the exception (*i* is an incrementally generated number).		`G ((a1 & ExceptionType_i = ExceptType_1)-> X a2)`
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 [1, p. 391]. Interrupting edges of a region have their source node in the region and their target node outside the region. A boolean condition, namely, "isInterrupted" is specified for handling the interruptions. When the event occurs the condition becomes true, and the action connected through interrupting_edge will be executed. Otherwise, negation of the constraint (i.e., !isInterrupted) is true and remaining activities will be executed.		1)`G (ac & isInterrupted -> X interrupting_edge)& G !(ac & interrupting_edge)` 2)`G (interrupting_edge -> X a2)` 3)`G (InitialNode & !( isInterrupted)-> F a1)`
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.		1)`(G (in_pn -> X a1)& G (a1 -> Y in_pn))` 2)`(G (an -> X out_pn)& G (out_pn -> Y an))`
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. To deal with the loop, a control variable, namely "max" is initialised with an initial value of zero. When the maximum iterations (max = 2) are reached, a new iteration cannot start and execution of the loop will terminate.	 [guard_1 & max >= 0 & max< 2] Merge Node a1 Decision Node [guard_2] [guard_3 | max = 2] a2	`(DecisionNode -> F a2)| ( DecisionNode & (max >= 0 & max < 2)-> F MergeNode)| ((DecisionNode & max = 2)-> F FlowFinal)& ! F( MergeNode)| F(a1)`

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 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.

Algorithm 1 Mapping UML Activity Diagram A into LTL Formulas

```
 1: procedure TRANSLATE($A$)
 2:     $Q \leftarrow \varnothing$                                      ▷ $Q$ is the queue of non-visited nodes
 3:     $V \leftarrow \varnothing$                                      ▷ $V$ is the queue of visited nodes
 4:     $Q \leftarrow Q \cup$ 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($n$)       ▷ for mapping of SMV descriptions we use generate_smv_code($n$)
 9:         $N_{outgoings} \leftarrow$ get_outgoing_nodes($n$)
10:         for all $m \in N_{outgoings}$ do
11:             if ($m \notin V$) then
12:                 $Q \leftarrow Q \cup \{m\}$
```

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. LTL-based transformation rules that constitutes the individual function `generate_ltl_code(`n`)` are presented in Table 2. The pair of triple apostrophes (''') denotes the string templates used for generating code in the our implementation based on Eclipse Xtend framework[2]. 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 all incoming flows have been received.

In our prior work [27], we introduced LTL-based transformation rules for a small basic set of activity diagram constructs including actions, sequences, fork, join, decision, and merge. The transformation rules presented in [27] 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: "$\mathbf{G}(a1 \rightarrow \mathbf{F}(b1 \wedge b2 \wedge \mathbf{F}(c1 \texttt{ xor } \mathbf{F}(d1 \wedge d2))))$". 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` [65]. Therefore, a `Decision Node` can be implemented using the "`xor`" operator or its equivalent but more complex form "$(a \wedge \neg b) \vee (\neg a \wedge b)$".

[2]See https://eclipse.org/xtend

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 $n \in initial\_node \mid n \in action$ do
5:         if $m \in N_{outgoings}$ then
6:             '''
7:                 LTLSPEC G(«$n$» -> F «$m$»)
8:             '''
9:     for all $n \in JoinNode \wedge i \in N_{incomings}$ do
10:        if $(i \geq 0) \wedge V \leftarrow V \cup \{i\}$ then
11:            '''
12:                LTLSPEC G((«$i$» & «$i$») -> F «$n$»)
13:            '''
```

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 "$(a1 \rightarrow \mathbf{F}a2)$" 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.

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 article. 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 [1, 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 >=0 & max < 2) leads to the merge node shown in Table 2.

We note that an LTL formula can be as simple as $\mathbf{G}(a1 \rightarrow \mathbf{F}a2)$ in case of representing the temporal relationship between two actions. Nevertheless, an LTL formula can also be quite complex, like $\mathbf{G}(((JoinNode \wedge \neg a1 \wedge \neg a2) \vee (\neg JoinNode \wedge a1 \wedge \neg a2) \vee (\neg JoinNode \wedge \neg a1 \wedge a2)) \rightarrow \mathbf{F}MergeNode)$ 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.

4.2. 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 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 [1]. On the other hand, 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). 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

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

```
procedure GENERATE_SMV_CODE($n$);
    extracts node information;
    binds input values and generates SMV descriptions using the following templates:
    '''
    VAR
        «n» :  boolean;                                      ▷ State variable declaration
    ASSIGN
        init(«n») := «node-initial-state»                    ▷ Definitions of state transitions
        next(«n») := case
            «incoming-guard-condition(s)» :  TRUE;
            «n» :  FALSE;
        esac;
    '''
```

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. 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 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 Xtend[3], 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`)`.

4.2.1. Dealing with Data

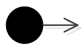
The UML activity diagram can contain variables such as integer, real, or string [1]. Thus, directly mapping variables of these types to NuSMV increases a finite state space which might lead to the state space infinite [20]. Nevertheless, we note that the constraints that are associated with nodes or edges can affect the behaviour of a UML activity diagram. The range of constraints in a UML activity diagram, regardless of their domains, is $\mathbb{B}$ = {`true`, `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 [20]. 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 LTL formula corresponding to a loop in Table 2. The predicates are also used to handle infinite state space [20]. 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 (see Sections 4.2.4, 4.2.5 and 4.2.6).

4.2.2. 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 2).

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



```
VAR
   «InitialNode» : boolean;
ASSIGN
   init(«InitialNode») := TRUE;
   next(«InitialNode») := case
         «InitialNode» : FALSE;
      esac;
```

(a) Initial Node (b) SMV Generation Rules for Initial Nodes

Figure 2: Translation of Initial Nodes into SMV Descriptions

4.2.3. 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* [1]. 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 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 10 in Algorithm 3). The node's state shall be switched back to false after the execution.

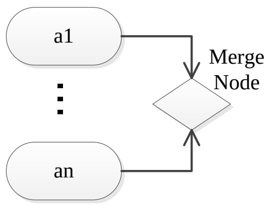
According to the UML 2 specification [1] the semantics of `Send Signal Action` and `Accept Event Action` inherit from an `Action`. Thus, we can implement the state transitions of `Send Signal Action` and `Accept Event Action` similar to that of an `Action` node. Note that an `Accept Event Action` can 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 `isEventOccur` initialises to `false` and is set to `true` when an event arrives.

```
VAR
   «node» : boolean;
ASSIGN
   init(«node») := FALSE;
   next(«node») := case
         «incoming_1» & «incoming_2» & ... & «incoming_n» : TRUE;
         «node» : FALSE;
      esac;
```

Figure 3: Generic Rules for Mapping UML Constructs to SMV Descriptions

4.2.4. Merge Node

A `Merge Node` brings together multiple alternative control flows and exclusively accepts one among them [1, 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 "$a1$ 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` [65]. This semantics does not precisely reflect the (semi-)formal description of `Merge Nodes` presented in the UML 2 specification [1, 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 [1, 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.



(a) Merge Node

```
 1 VAR
 2    «a1» : boolean;
 3    ...
 4    «an» : boolean;
 5    «MergeNode»: boolean;
 6    -- temporary variable
 7    «merge_flag_i» : {undetermined, «in_a1»,...,«in_an»}
 8 ASSIGN
 9    ... -- the initializations and transitions of a1...an are omitted
10    init(«merge_flag_i») := undetermined;
11    next(«merge_flag_i»):= case
12          («merge_flag_i» = undetermined) & («a1» | ... | «an»): {«in_a1»,...,
                 «in_an»};
13          TRUE : undetermined;
14       esac;
15    init(«MergeNode») := FALSE;
16    next(«MergeNode») := case
17          «a1» & !«a2» & ... & !«an» : TRUE; -- only a1 is true
18          !«a1» & «a2» & ... & !«an» : TRUE; -- only a2 is true
19          ...
20          !«a1» & !«a2» & ... & «an» : TRUE; -- only an is true
21          «merge_flag_i» = «in_a1» | ... |«merge_flag_i» = «in_an» : TRUE;
22          «MergeNode» : FALSE;
23       esac;
```

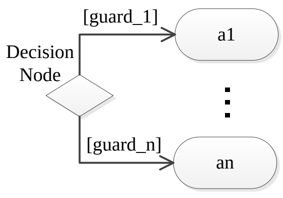
(b) SMV Generation Rules for Merge Nodes

Figure 4: Translation of Merge Nodes into SMV Descriptions

In this article, 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 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_a`x" where $x = 1,\ldots,n$—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 $a1\ldots an$ 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 are activated, NuSMV will bind `merge_flag_i` to a certain value "`in_a`x" in the first place, to "`undetermined`" in the next transition, then to another value "`in_a`y" 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_a`x", where $x = 1,\ldots,n$.

4.2.5. 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 [1, 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.



(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;
```

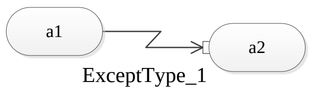
(b) SMV Generation Rules for Decision Nodes

Figure 5: Translation of Decision Nodes into SMV Descriptions

Figure 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 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.

4.2.6. *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 [1, p. 374]. However, theory proposed in the standard does not clearly define how to declare the type of the exception or its



(a) Exception Handler

```
1  VAR
2     «a1» : boolean;
3     «a2» : boolean;
4     «ExceptionType_i»: {undetermined, «no_Exception», «ExceptType_1»
          ,..., «ExceptType_n»};
5  ASSIGN
6     init(«a1») := FALSE;
7     next(«a1») := case
8           «incoming_1» & «incoming_2» & ... & «incoming_n» : TRUE;
9           «ExceptionType_i» = «no_Exception» : FALSE;
10          «ExceptionType_i» = «Exception_1» : FALSE;
11          ...
12          «ExceptionType_i» = «Exception_n» : FALSE;
13          «a1» : FALSE;
14       esac;
15    init(«ExceptionType_i») := undetermined;
16    next(«ExceptionType_i») := case
17          («ExceptionType_i» = undetermined) & «a1» : {«no_Exception»,
               «ExceptType_1»,..., «ExceptType_n»};
18          TRUE : undetermined;
19       esac;
20    --the handler body
21    init(«a2») := FALSE;
22    next(«a2») := case
23          «ExceptionType_i» = «ExceptType_1» : TRUE;
24          «a2»  : FALSE;
25       esac;
```

(b) SMV Generation Rules for Exception Handlers

Figure 6: Translation of Exception Handlers into SMV Descriptions

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.

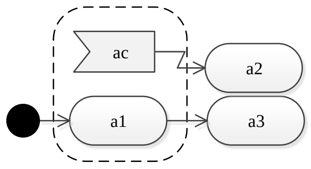
Figure 6 illustrates the rules for mapping an `Exception Handler` into SMV descriptions. For handling the exception, we introduce a temporal variable, namely, `ExceptionType_i` (*i* is an incrementally generated number) for more than one handlers connected to the protected node. `ExceptionType_i` has 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` (*a*2) starts when `ExceptType_1` matches with a exception raised by a protected node (Line 23).

4.2.7. 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 [1, 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 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 *a*2 outside the region (Line 20–23). If `isInterrupted` is `false` (i.e., negation of `isInterrupted` is `True`), then all

```
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;
```



(a) Interruptible Activity Region

(b) SMV Generation Rules for Interruptible Activity Region

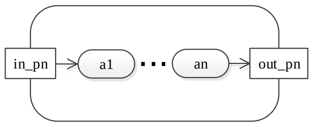
Figure 7: Translation of Interruptible Activity Region into SMV Descriptions

the nodes within the region complete their execution (Line 25–33) other than the source and target nodes of the *interrupting_edge*.

4.2.8. 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 [1, p. 346].

`Activity Parameter Nodes` are abstracted as boolean variables (Line 2–3). As mentioned in section 4.2.1 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 $x = 1,\ldots,n$—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` 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 8.



(a) Activity Parameter Node

```
VAR
   «in_pn» : boolean;
   «out_pn»: boolean;
   -- temporary variables
   «in_par» : {undetermined, «in_val1»,..., «in_valn»};
   «out_par»: {undetermined, «out_val1»,..., «out_valn»};
ASSIGN
   init(«in_par») := undetermined;
   next(«in_par») := case
         («in_par» = undetermined) : {«in_val1»,..., «in_valn»};
         TRUE : undetermined;
      esac;
   -- the input parameter node
   init(«in_pn») := FALSE;
   next(«in_pn») := case
         «in_par» = «in_val1» & ... & «in_par» = «in_valn» : TRUE;
         «in_pn» : FALSE;
      esac;
   ...
   init(«out_par») := undetermined;
   next(«out_par») := case
         («out_par» = undetermined) : {«out_val1»,..., «out_valn»};
         TRUE : undetermined;
      esac;
   -- the output parameter node
   init(«out_pn») := FALSE;
   next(«out_pn») := case
         «out_par» = «out_val1» & ... & «out_par» = «out_valn» : TRUE;
         «out_pn» : FALSE;
      esac;
```

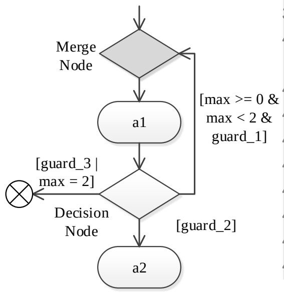
(b) SMV Generation Rules for Activity Parameter Nodes

Figure 8: Translation of Activity Parameter Nodes into SMV Descriptions

4.2.9. 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 9 as inspired by [20, 55]. Eshuis and Wieringa use strong fairness (also known as compassion) constraints to exit loops eventually [20], whilst Guelfi and Mammar only consider loops with predefined limited iterations [55].

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 *max >=0 & max < 2* and *max = 2* are shown in Figure 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*, which is 0 initially (i.e., equal to minimum iteration). Its value is incremented each time the action $a1$ is executed (Line 22–25). If *max* become equals to maximum number, a new iteration cannot start and execution of the loop will terminate. The transformation rule for loops is shown in Figure 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 Sections 4.2.5 and 4.2.4, respectively. As mentioned in Section 4.1, the test and setup sections of a UML 2 `Loop Node` are similar to the decision condition and the incoming (e.g., action) of



(a) Loop Structure

```
VAR
      «MergeNode»: boolean;
      «DecsisonNode»: boolean;
      «a1» : boolean;
      ...
      «max» : «0..2»; -- control variable
      «merge_flag_i» : {undetermined, «in_a1», «in_guard_1»};
      «post_decision_i» : {undetermined, «guard_1», «guard_2», «guard_3»};
ASSIGN
      init(«merge_flag_i») := undetermined;
      next(«merge_flag_i»):= case
            («merge_flag_i» = undetermined) & («incoming_1» | «DecsisonNode»
                 ): {«in_incoming_1»,«in_guard_1»};
            TRUE : undetermined;
         esac;
      init(«MergeNode») := FALSE;
      next(«MergeNode») := case
            «incoming_1» & !«DecsisonNode» : TRUE;
            !«incoming_1» & «DecsisonNode» : TRUE;
            «merge_flag_i» = «in_incoming_1» | «merge_flag_i» = «in_guard_1»
                   : TRUE;
            «MergeNode» : FALSE;
         esac;
      init(«a1») := FALSE;
      next(«a1») := case
            «MergeNode» : TRUE;
            «a1»    : FALSE;
        esac;
      init(«DecisionNode») := FALSE;
      next(«DecisionNode») := case
            «a1» :TRUE;
            «DecisionNode» : FALSE;
         esac;
      init(«max»):= (0);
      next(«max»):= case
             («max» >= 0) & («max» < 2) : «max» +1;
             («max» = 2) : «max»;
             TRUE : «max»;
          esac;
      init(«post_decision_i») := undetermined;
      next(«post_decision_i») := case
            «DecisionNode» & («post_decision_i» = undetermined) : {guard_1,
                 guard_2, guard_3};
            TRUE : undetermined;
         esac;
      .....
      init(«FlowFinal») := FALSE;
      next(«FlowFinal») := case
            «post_decision_i» = «guard_3» | «max» = 2 : TRUE;
            «FlowFinal» : FALSE;
         esac;
```

(b) SMV Generation Rules for Loops

Figure 9: Translation of Loops into SMV Descriptions

the merge node. Therefore, aforementioned mapping rules for the loop can be applied on the `Loop Node`.

4.3. *Step 3: Containment Checking and Dealing with Containment Inconsistencies*

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. Unfortunately, it is often difficult for the developers who have limited knowledge of the underlying formal methods to comprehend the counterexamples that may contain numerous information such as states numbers, input variables over tens of cycles and internal transitions, and so on [18, 19]. The developers might have to devote considerable time analysing an error trace in order to understand the causes of inconsistencies. Understanding the causes of containment violations is a prerequisite to resolve the inconsistencies in the models.

To alleviate these issues, an efficient analysis of the generated counterexample is supported in our proposed containment checking approach. 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. The analysis of counterexample in our proposed 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.

4.3.1. Locating Causes of Containment Inconsistencies

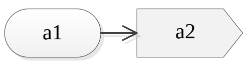
In order to locate the causes of containment inconsistencies, the output trace file is scrutinised and parsed to determine the unsatisfied LTL formulas. The extracted formulas and SMV descriptions together with LTL-based transformation rules are traversed to find out why the elements of the high-level activity diagram are not matched by their corresponding low-level counterparts. This is performed in two steps: Firstly, the counterexample analyser verifies whether all the elements (e.g., actions, control nodes and edges/guards) that exist in the high-level activity diagram are also present in the low-level activity diagram by using the function match() (see Algorithm 4). Note that match() essentially compares the nodes' names, types, and/or guard conditions. For this, the missing element cause (either one, multiple, or all elements could be missing) is detected and the countermeasure (i.e., insert the missing element at a particular position in the low-level model) is suggested. Secondly, the rules related to unsatisfied LTL formulas for different possible kinds of elements in the activity diagram are evaluated. For this, the exact position of the corresponding elements in the high-level model related to unsatisfied LTL formulas are matched with elements in the low-level model. In particular, the sequence (of elements of the low-level model) from the SMV descriptions is scrutinised and corresponding element (e.g., action, control node and so on) causing the violation of the LTL formulas is located. The 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 5.

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 10b), `Send Signal Action` ($a2$) is not preceded by Action $a1$ as compared to the high-level diagram (shown in Figure 10a). In this case, the results produced by NuSMV indicate that `G (a1 -> X a2)` 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

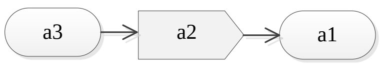
Algorithm 4 Verifying Missing/Deleted Elements in the Low-Level Activity Diagram

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

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 $a2$.



(a) The High-Level Activity Diagram



(b) The Low-Level Activity Diagram

Figure 10: Send Signal Action Preceded by Different Action

Algorithm 5 Verify Misplacement of Elements in the Low-Level Activity Diagram

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

Table 3 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 3, a state sequence is denoted by $k-1, k, k+1, ..., kn$, 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 $a1, ..., an$ and $b1, ..., bn$ will be parameterized with the actual node names.

Table 3: Tracking Back the Causes of Violation Due to Misplacement of Elements and Possible Countermeasures

Unsatisfied Rule	Causes of Violation Due to Misplacement	Possible Countermeasure(s)
`G (a1 -> 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.	• 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.

(continued on next page)

Table 3: Tracking Back the Causes of Violation Due to Misplacement of Elements and Possible Countermeasures

Unsatisfied Rule	Causes of Violation Due to Misplacement	Possible Countermeasure(s)
`G (ForkNode -> F (a1 & ... & an))& G ((a1 & ... & an)-> O ForkNode)`	The Fork Node rule is violated because Fork Node is not followed by the elements.	• Add elements $(a1...an)$ in the low-level model after the particular Fork Node. • Elements $(a1...an)$ must exist after the particular Fork Node.
`G ((a1 &...& an)-> F JoinNode)`	The Join Node rule is violated when elements $(a1...an)$ exist as succeeding elements of a Join Node, but are not followed by a Join Node.	• Elements $(a1...an)$ shall be followed by a particular Join Node. • Remove element(s) $(a1...an)$ from the low-level model. • Replace flawed elements(s) $(b1...bn)$ with the corresponding correct elements $(a1...an)$.
`(DecisionNode -> F ((a1 & !a2)| (!a1 & a2)))`	The Decision Node rule is violated, when both of the branches return either `true` or `false`. It means that the Decision Node is not followed by branches/elements (i.e., $a1$ and $a2$) in the low-level model.	• Replace the flawed branches (e.g., $b1$ and $b2$) with the corresponding correct branches $a1$ and $a2$ after the particular Decision Node. • Remove flawed branches ($b1$ and $b2$) from the low-level model, respectively. • Put corresponding Decision Node before $a1$ and $a2$.
`G (((a1 & !a2)| (!a1 & a2))-> F MergeNode)`	The Merge Node rule is unsatisfied, because elements (i.e., $a1$ and $a2$) 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.	• Put the particular Merge Node after $a1$ and $a2$ in the model. • Replace the flawed elements $b1$ and/or $b2$ with corresponding correct elements $a1$ and $a2$ before the particular Merge Node.
`G (a1 -> X a2)`	The Send Signal Action rule is violated, if Send Signal Action $a2$ either exists in the low-level model as an eventually succeeding element of Action $a1$ (i.e., exits at $(k+2)$ to (kn), but not at $(k+1)$ next element of the $a1$) or preceding element of the $a1$ (i.e., exits at $(k-1)$ to $(k-n)$).	• Add $a2$ immediately after $a1$ in the low-level model. • Swap the occurrence of $a1$ and $a2$.
`1)G (a1 -> X a2)& G !(a1 & a2)` `2)G (a1 -> G (a2 -> X a3))`	Accept Event Action $a2$ (transitively) exists in the low-level model as a preceding element of Action $a1$ but not after $a1$. In the second rule, the Action $a3$ is not at next position $k+1$ (i.e., $a3$ exists as an eventually succeeding element of $a2$ $(k+2)$ to (kn)).	• Replace the flawed action $b1$ with the corresponding correct action ($a1$ or $a3$). • Put $a2$ after $a1$ in the low-level model. • Put $a3$ immediately after $a2$ in the low-level model, where $a1...a3$ and $b1$ are action nodes that will be parameterized with the actual action names.
`G ((a1 & ExceptionType_i = ExceptType_1)-> X a2)`	The rule is violated, because $a2$ (i.e., exception handler) does not exist after $a1$ (i.e., protected node) and Exception Type.	• Put the correct handler body $a2$ after the corresponding protected node and Exception Type in the low-level model.
`1)G (ac & isInterrupted -> X interrupting_edge )& G !(ac & interrupting_edge)` `2)G (interrupting_edge -> X a2)` `3)G (InitialNode & ! ( isInterrupted)-> F a1)`	The first rule is unsatisfied, because the element $interrupting\_edge$ is not at next position $k+1$ of Accept Event Action ac (i.e., it exists as an eventually succeeding element of ac $(k+2)$ to (kn)). The second rule is violated, because $interrupting\_edge$ and/or $a2$ are misplaced. The third rule is violated, because $a1$ is misplaced.	• Put $interrupting\_edge$ immediately after Accept Event Action ac. • Remove $interrupting\_edge$ and $a2$ from the low-level model. • Remove element $a1$ from the low-level model. • Replace flawed elements $b1...b3$ with corresponding correct elements $interrupting\_edge$, $a1$ and $a2$.
`1)(G (in_pn -> X a1)& G (a1 -> Y in_pn))` `2)(G (an -> X out_pn)& G (out_pn -> Y an))`	The first rule is violated, because Action $a1$ (transitively) exists in the low-level model as an eventually succeeding element of input Activity Parameter Node in_pn, but not as next element of the in_pn. The second rule is violated, because the output Activity Parameter Node out_pn (transitively) exists in the low-level model as a preceding element of Action an, but not as next element of an.	• Put $a1$ immediately after in_pn in the low-level model. • Put out_pn immediately after an in the low-level model.
`(DecisionNode -> F a2)| (DecisionNode & (max >= 0 & max < 2)-> F MergeNode)| ((DecisionNode & max = 2) -> F FlowFinal)& ! F(MergeNode)| F(a1)`	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 $a2$ exists before a DecisionNode, but not followed by a DecisionNode. The MergeNode does not exist before Action $a1$.	• Replace flawed elements (action, decision node, or merge node) with the corresponding correct elements, such as $a1$ and $a2$. • Put action $a1$ before a Decision Node. • Put action $a2$ after a Decision Node. • Add corresponding MergeNode and $a1$ in the low-level model. • Add the flow final in the low-level model.

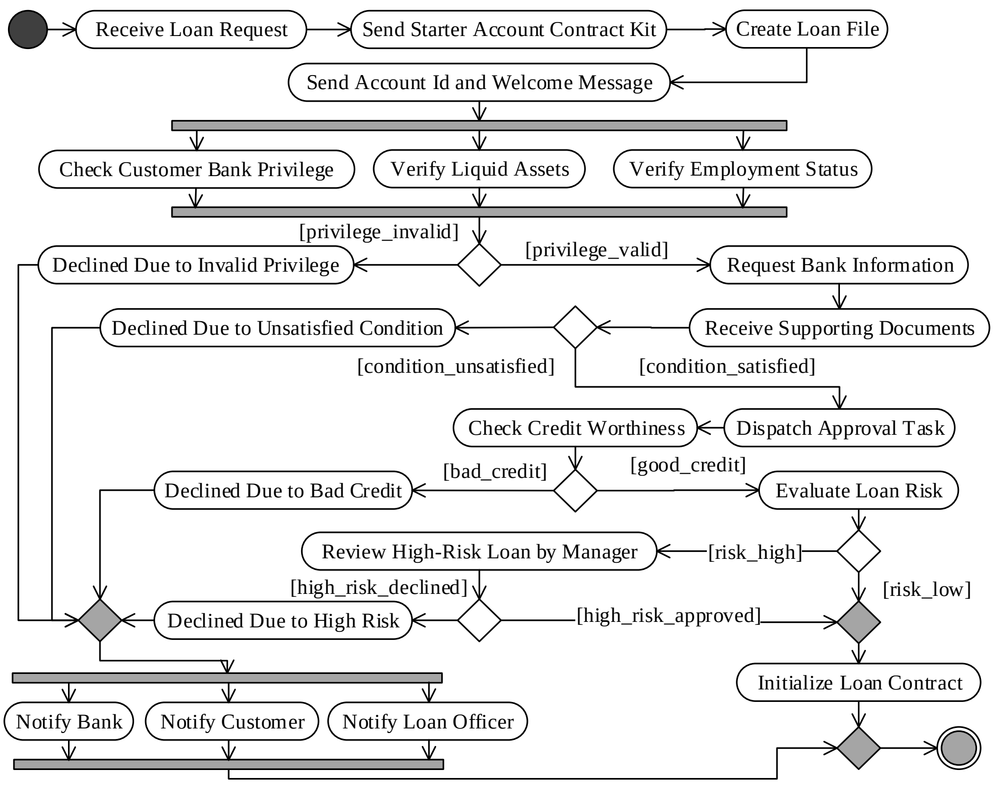


Figure 11: High-Level Activity Diagram of Loan Approval System

4.3.2. Visualisation of Containment Causes and Relevant Countermeasures

In this section, we explain how the discordance between high-level model and its low-level counterparts are presented to the developers in a user friendlier manner in comparison to the generated counterexamples. We opt to implement the visualisation in Eclipse Papyrus[4], which is an Eclipse based open source UML 2 tool. Nevertheless, the same visualisation technique can be integrated with other UML tools given the input from the counterexample analyser. The visual representation is based on the information produced in the aforementioned step along with the low-level model as input. In particular, the description of the cause(s) along with the relevant potential countermeasures to address the containment inconsistency is annotated with the UML construct whose corresponding LTL formula is violated, for instance, fork node, merge node and accept event action. In order to enhance the visibility and understandability of the counterexample analysis results, the elements related to unsatisfied LTL formulas are also visually presented. For instance, the UML construct that corresponds to the violated LTL formula is highlighted in blue whilst the elements responsible for causing the containment inconsistencies are highlighted in red, and the elements that satisfied the corresponding LTL formula appear in green. After locating the causes of containment inconsistencies, the low-level activity diagram is updated based on the countermeasures and re-mapped to its formal description, and then will be re-verified. This process iterates until no more containment inconsistencies are detected. To differentiate more than one unsatisfied formula, the elements areas are highlighted with shades of the particular colour making areas of interest better visible.

5. 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

[4]See https://www.eclipse.org/papyrus

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 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.

Table 4: LTL Formulas Generated from the High-Level Loan Approval Activity Diagram

UML Constructs	**Generated LTL Formulas**

Sequence	`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);`

Fork Node	`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);`

Join Node	`LTLSPEC G ((CheckCustomerBankPrivilege & VerifyLiquidAssets & VerifyEmploymentStatus) -> F JoinNode);`
	`LTLSPEC G ((NotifyBank & NotifyCustomer & NotifyLoanOfficer) -> F JoinNode2);`

Decision Node	`LTLSPEC (DecisionNode -> F (DeclinedDuetoInvalidPrivilege xor RequestBankInformation));`
	`LTLSPEC (DecisionNode1 -> F (DeclinedDuetoUnsatisfiedCondition xor DispatchApprovalTask));`
	`LTLSPEC (DecisionNode2 -> F (DeclinedDuetoBadCredit xor EvaluateLoanRisk));`
	`LTLSPEC (DecisionNode3 -> F (ReviewHigh_RiskLoanbyManager xor MergeNode));`
	`LTLSPEC (DecisionNode4 -> F (DeclinedDuetoHighRisk xor MergeNode));`

Merge Node	`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);`

5.1. 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 11. 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 2. For instance, the LTL formula "`LTLSPEC (DecisionNode -> F (DeclinedDuetoInvalidPrivilege xor RequestBankInformation))`" is generated for the Decision Node (as discussed in Section 4.1). 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 4 shows the generated LTL formulas from the high-level loan approval system.

5.2. 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 is automatically converted into SMV descriptions using our aforementioned transformation rules.

Listing 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 4.2. 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.

5.3. Containment Checking Results

The containment checking is achieved by using the NuSMV model checker to check the generated SMV descriptions (as illustrated in Listing 1) against the LTL formulas generated from the high-level model (as presented in Table 4). The LTL formulas and SMV descriptions are combined into one NuSMV input file and executed by the NuSMV model checker. The NuSMV performs the containment checking on the semantics of activity diagram by exploring the reachable state-space of a model. However, the syntactic analysis considers that the specification of an activity diagram conforms to the abstract syntax specified by the metamodel. In particular, it only ensures that identifier names used in the high-level diagram must be defined in the low-level diagram. The semantics of elements such as fork node rule violation cannot be identified by syntactic analysis. Therefore, NuSMV Model checker is used for the containment checking. Listing 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`. In particular, three LTL formulas of Table 4 are false out of twenty-seven generated formulas. 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. 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, discussed in Section 4.3. 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

```
MODULE main
 VAR
        InitialNode   : boolean;
        ReceiveLoanRequest : boolean;
        ReviewRequest  : boolean;
        GenerateAccountId : boolean;
        ForkNode   : boolean;
        JoinNode   : boolean;
        MergeNode    : boolean;
        merge_flag_629    : {undetermined, in_DecisionNode4, in_DecisionNode3};
...
  ASSIGN
--
 init(InitialNode) := TRUE;
 next(InitialNode) := case
      InitialNode : FALSE;
      TRUE : InitialNode;
   esac;
 init(ReceiveLoanRequest) := FALSE;
 next(ReceiveLoanRequest) := case
      InitialNode : TRUE;
      ReceiveLoanRequest : FALSE;
      TRUE : ReceiveLoanRequest;
   esac;
...
 init(SendStarterAccountContractKit) := FALSE;
 next(SendStarterAccountContractKit) := case
      GenerateAccountId : TRUE;
      SendStarterAccountContractKit : FALSE;
      TRUE : SendStarterAccountContractKit;
   esac;
 init(SendAccountIdandWelcomeMessage) := FALSE;
 next(SendAccountIdandWelcomeMessage) := case
      SendStarterAccountContractKit : TRUE;
      SendAccountIdandWelcomeMessage : FALSE;
      TRUE : SendAccountIdandWelcomeMessage;
   esac;
 ...
 init(post_decision_975) := undetermined;
 next(post_decision_975) := case
      DecisionNode & (post_decision_975 = undetermined) : {guard_1, guard_2};
      TRUE: undetermined;
   esac;
 init(DeclinedDuetoInvalidPrivilege) := FALSE;
 next(DeclinedDuetoInvalidPrivilege) := case
      post_decision_975 = guard_1 : TRUE;
      DeclinedDuetoInvalidPrivilege : FALSE;
      TRUE: DeclinedDuetoInvalidPrivilege;
   esac;
 init(RequestBankInformation) := FALSE;
 next(RequestBankInformation) := case
      post_decision_975 = guard_2 : TRUE;
      RequestBankInformation : FALSE;
   TRUE : RequestBankInformation;
   esac;
 init(CreateLoanFile) := FALSE;
 next(CreateLoanFile) := case
      RequestBankInformation : TRUE;
      CreateLoanFile : FALSE;
   TRUE : CreateLoanFile;
 esac;
 ...
```

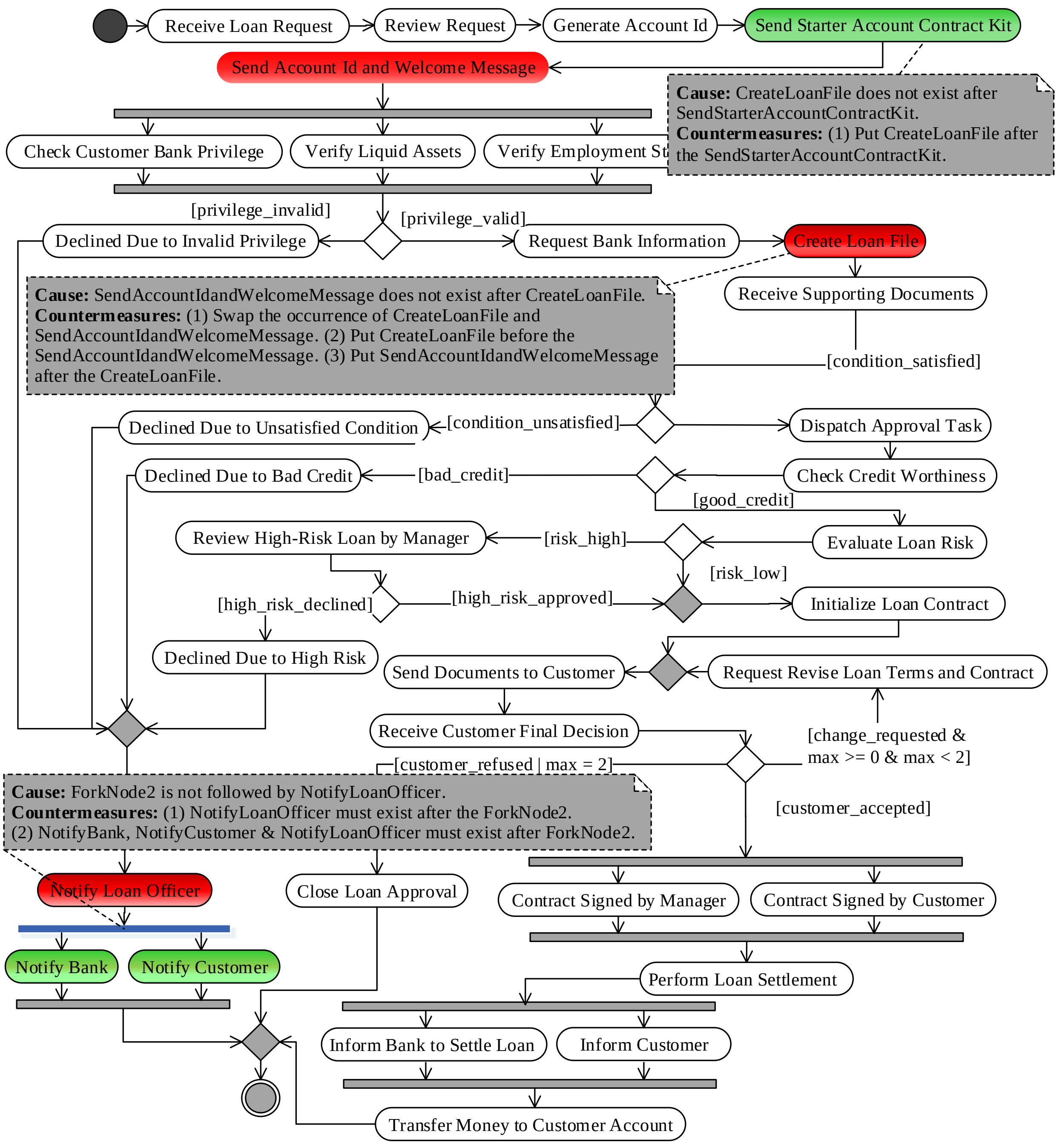
Listing 1: Excerpts of the SMV Description for the Low-level Loan Approval Model

```
$ 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
ReviewRequest = FALSE
GenerateAccountId = FALSE
SendStarterAccountContractKit = FALSE
...
-> State: 1.20 <-
merge_flag_322 = undetermined
-- specification G (CreateLoanFile -> F SendAccountIdandWelcomeMessage) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
InitialNode = TRUE
ReceiveLoanRequest = FALSE
...
-- specification G (MergeNode1 -> F ForkNode2) is true
-- specification ( G (ForkNode2 -> F ((NotifyBank & NotifyCustomer) & NotifyLoanOfficer)) & G (((NotifyBank
    & NotifyCustomer) & NotifyLoanOfficer) -> O ForkNode2)) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 3.1 <-
InitialNode = TRUE
ReceiveLoanRequest = FALSE
...
```

Listing 2: NuSMV Containment Checking Result along with a Counterexample

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 `SendStarterAccountContractKit` action of the formula is highlighted in green colour. As we can see in Figure 12, 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 (ForkNode2 -> F (NotifyBank & NotifyCustomer & NotifyLoanOfficer))& G ((NotifyBank & NotifyCustomer & NotifyLoanOfficer)-> O 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 12, 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. Without the counterexample analysis, users 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 UML model, and then locate the corresponding inconsistency within the UML model, meaning that the complex matching between the variables and states in the counterexample and the elements of the activity diagrams must be performed manually.


Receive Loan Request
Review Request
Generate Account Id
Send Starter Account Contract Kit
Send Account Id and Welcome Message
Check Customer Bank Privilege
Verify Liquid Assets
Verify Employment St
[privilege_invalid]
[privilege_valid]
Declined Due to Invalid Privilege
Request Bank Information
Create Loan File
Receive Supporting Documents
[condition_satisfied]
Declined Due to Unsatisfied Condition
[condition_unsatisfied]
Dispatch Approval Task
Declined Due to Bad Credit
[bad_credit]
Check Credit Worthiness
[good_credit]
Review High-Risk Loan by Manager
[risk_high]
Evaluate Loan Risk
[risk_low]
[high_risk_declined]
[high_risk_approved]
Initialize Loan Contract
Declined Due to High Risk
Send Documents to Customer
Request Revise Loan Terms and Contract
Receive Customer Final Decision
[change_requested & max >= 0 & max < 2]
[customer_refused | max = 2]
[customer_accepted]
Notify Loan Officer
Close Loan Approval
Contract Signed by Manager
Contract Signed by Customer
Notify Bank
Notify Customer
Perform Loan Settlement
Inform Bank to Settle Loan
Inform Customer
Transfer Money to Customer Account

Cause: CreateLoanFile does not exist after SendStarterAccountContractKit.
Countermeasures: (1) Put CreateLoanFile after the SendStarterAccountContractKit.

Cause: SendAccountIdandWelcomeMessage does not exist after CreateLoanFile.
Countermeasures: (1) Swap the occurrence of CreateLoanFile and SendAccountIdandWelcomeMessage. (2) Put CreateLoanFile before the SendAccountIdandWelcomeMessage. (3) Put SendAccountIdandWelcomeMessage after the CreateLoanFile.

Cause: ForkNode2 is not followed by NotifyLoanOfficer.
Countermeasures: (1) NotifyLoanOfficer must exist after the ForkNode2.
(2) NotifyBank, NotifyCustomer & NotifyLoanOfficer must exist after ForkNode2.

Figure 12: Low-Level Activity Diagram of Loan Approval System After Running Counterexample Analysis

6. 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 [26]. It might be noted that multiple interleaving and iterative activities have been covered in the particular scenarios.

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, CRM fulfilment system [66] and billing renewal system [67]. 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 (CRM) 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. Due to the space limitation, we omit the detail of these scenarios. 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 Xtend[5]. Note that we used the NuSMV model checker version 2.5.4 for verifying the containment relationship.

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 5 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 6 shows the containment checking time, total time (i.e., verification plus loading and translation time) and violated formulas.

Table 5: Model Size and Translation Time

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	20	23	53
Action Nodes	7	13	12	16	14	22	21	33	29	92
Edges	21	35	23	30	35	54	43	66	70	179
Total Elements	40	62	42	54	61	93	78	119	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	0.646±0.22	0.411±0.04	0.460±0.12	3.115±0.41	0.697±0.15	4.798±0.31

The evaluation results show that the containment checking time spent by NuSMV for the loan approval is longer than for the itinerary management, CRM fulfilment 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. Three out of twenty-seven LTL formulas specified in Table 4 are not satisfied. Specifically, the fork node rule and sequential rules are violated in the loan approval system. The causes of containment violations and their resolutions are explained in the case study (Section 5).

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 Model (HL=122 and LL=324) is

[5]See http://www.eclipse.org/xtend

around 35630 ms (i.e., about 0.59 minutes). The model checking time, although it grows for such a large number of elements, is still reasonable in the context of a typical working environment. In the Synthetic Larger Model, the containment relationship is not satisfied for two out of thirty-six LTL formulas; the fork node and decision node rules are violated. Likewise, the causes of the violations and appropriate guidelines to resolve the particular violations might be identified based on the counterexample analysis method described in Section 4.3. 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 lager model, the low-level activity diagrams of these models are updated based on the countermeasures 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.

Table 6: Performance Evaluation Results

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	35 630.958
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 (2ˆ6.61471)	40 (2ˆ5.32193)	221 (2ˆ7.7879)	$5.6776e+006$ (2ˆ22.4369)	931 (2ˆ9.86264)
Total States	$2.37763e+013$ (2ˆ44.4346)	$3.71085e+012$ (2ˆ41.7549)	$2.00386e+015$ (2ˆ50.8317)	$1.89108e+022$ (2ˆ74.0016)	$5.68419e+036$ (2ˆ122.096)

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 [59, 68]. This problem can be partially alleviated by using predicates instead of enumeration types [20]. 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 [53]. In this work constraints are abstracted and encoded as boolean representatives (see Section 4.2). This encoding decision can help reducing significantly the state space under consideration. The translation in our previous work consumes approximately three seconds for verification of the loan approval system. Our improvements reduced the verification time of the loan approval system from three seconds to one and half a second (see Table 6).

7. 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:

RQ1: How to perform automated transformation of behaviour models into formal specifications and descriptions?

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 article, 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 $\mathscr{A}$ is a tuple (N,E,G) as mentioned in Section 3.3. 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 4.2 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 $(s_0, s_1, .., s_n)$ 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. A simple transition, in an activity diagram, from an action or an activity a_i to another action or activity $a_i + 1$, is formalised as (a_i -> $a_i + 1$), i.e., action a_i leads to action $a_i + 1$. In the case, we need to be sure that every execution path going through a_i also goes through $a_i + 1$. An LTL formula φ is true in state s if and only if φ holds for all paths starting in s. $s \models \varphi \equiv \forall \pi \in \text{Paths}(s).\pi \models \varphi$, where $\models$ is the satisfaction relation for formulas. For a finte state machine, a formula φ is valid for state machine M if and only φ holds in all the initial states of M. This means that φ holds in all paths starting from any initial state $M \models \varphi \equiv \forall s_0 \in I.s_0 \models \varphi$.

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 [53]. 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 SPIN. 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.

RQ2: How can we effectively communicate the containment checking results to the developers?

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 stakeholders are expected to update the low-level activity diagram based on the produced countermeasures; after that, the formal descriptions from updated diagram are generated, and containment checking is performed using NuSMV.

RQ3: 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 [66, 67] 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. It is decided to perform performance evaluation to check whether the approach is applicable to very large models and feasible for situations in which immediate results are needed. The evaluation results show that even with the input models having about 300 model elements, the verification time stays within less than 1 minute. Several interleaving and iterative activities have been contained in the industrial cases and synthetic larger model. In summary, 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 combing 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 [1]. 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 [69]. Nevertheless, Dwyer et al. [69] 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 [70].

8. Conclusion and Final Remarks

In this article, we have 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. 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.

The containment checking relationship between high-level and low-level activity diagrams can be verified by using existing model checking tools. However, the model checking techniques do not produce concrete and comprehensive information about the violation of the containment relationship and the problem of finding the cause of containment inconsistencies is usually delegated to the developers. In this context, we introduce our counterexample interpretation approach to analyse counterexamples, understand the cause of inconsistencies and provide developers with supporting evidences for resolving the cause of inconsistencies between high-level and low-level behaviour models. Therefore, the strong knowledge of formal methods is not required. To illustrate the applicability of the proposed approach, we realized five different use cases. Four of them are taken from the industrial scenarios and one is the Synthetic Larger Model. The performance evaluation is also carried out in those 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.

The fundamental constructs of the behaviour models described in this article are also supported in different other behaviour models such as BPMN and BPEL that are widely used for describing behaviours of process-centric information systems. As a result, our approach can be easily adapted to support the other behaviour models. For other types of behaviour models such as UML state machines, communication diagrams, and EPC that have significantly different constructs, the proposed transformation rules cannot be used straightforwardly. However, with extra efforts in defining transformation rules for these behaviour models, we may gain the same benefit from the automated generations of formal specification and properties as presented in this article. At the current level of development, our work has not incorporated the automated transformation of hierarchical model compositions of the input behaviour models. In the future we plan to investigate an efficient way to support such hierarchical model compositions. Another direction for future work is to extend our approach for other UML behaviour models such as statecharts.

Acknowledgement

The research leading to the results presented in this article is supported by the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001.

References

[1] Object Management Group (OMG), UML 2.4.1 Superstructure Specification, http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF, (Last accessed: June 2, 2018) (2011).

[2] Object Management Group (OMG), Business Process Model and Notation, http://www.omg.org/spec/BPMN/2.0, (Last accessed: May 20, 2018) (2011).

[3] A.-W. Scheer, ARIS — Vom Geschäftsprozess zum Anwendungssystem, 4th Edition, Springer, 2002.

[4] G. Spanoudakis, A. Zisman, Handbook of Software Engineering and Knowledge Engineering, World Scientific, 2001, Ch. Inconsistency management in software engineering: Survey and open research issues, pp. 329–380.

[5] F. J. Lucas, F. Molina, A. Toval, A systematic review of UML model consistency management, Information and Software Technology 51 (2009) 1631–1645. doi:10.1016/j.infsof.2009.04.009.

[6] F. UL Muram, H. Tran, U. Zdun, Systematic review of software behavioral model consistency checking, ACM Comput. Surv. 50 (2) (2017) 1–39. doi:10.1145/3037755.

[7] A. Tsiolakis, H. Ehrig, Consistency analysis of UML class and sequence diagrams using attributed graph grammars, in: Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Open Publishing Association, 2000, pp. 77–86.

[8] H. Eshuis, R. Wieringa, A formal semantics for UML activity diagrams - Formalising workflow models, CTIT technical report series, Centre for Telematics and Information Technology, University of Twente, Enschede, 2001.

[9] W. Yeung, Checking consistency between uml class and state models based on csp and b, J. UCS 10 (11) (2004) 1540–1559. doi:10.3217/jucs-010-11-1540.

[10] H. Wang, T. Feng, J. Zhang, K. Zhang, Consistency check between behaviour models, in: IEEE International Symposium on Communications and Information Technology, IEEE Computer Society, 2005, pp. 486–489. doi:10.1109/ISCIT.2005.1566899.

[11] T. Schäfer, A. Knapp, S. Merz, Model checking UML state machines and collaborations, Electronic Notes in Theoretical Computer Science 55 (3) (2001) 357 – 369, workshop on Software Model Checking (in connection with CAV'01). doi:10.1016/S1571-0661(04)00262-2.

[12] A. Knapp, S. Merz, C. Rauh, Model checking - timed UML state machines and collaborations, in: 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2 (FTRTFT), Springer-Verlag, 2002, pp. 395–416. doi:10.1007/3-540-45739-9_23.

[13] V. S. W. Lam, J. Padget, Consistency checking of sequence diagrams and statechart diagrams using the pi-calculus, in: 5th International Conference on Integrated Formal Methods, IFM'05, Springer-Verlag, 2005, pp. 347–365. doi:10.1007/11589976_20.

[14] F. UL Muram, H. Tran, U. Zdun, A model checking based approach for containment checking of uml sequence diagrams, in: 23rd Asia-Pacific Software Engineering Conference (APSEC), APSEC '16, IEEE Computer Society, Hamilton, New Zealand, 2016, pp. 73–80. doi:10.1109/APSEC.2016.021.

[15] F. UL Muram, H. Tran, U. Zdun, Towards containment checking of behaviour in architectural patterns, in: 22Nd European Conference on Pattern Languages of Programs, EuroPLoP '17, ACM, New York, NY, USA, 2017, pp. 29:1–29:19. doi:10.1145/3147704.3147736.

[16] F. U. Muram, M. A. Javed, H. Tran, U. Zdun, Towards a framework for detecting containment violations in service choreography, in: Proceedings of the IEEE International Conference on Services Computing, SCC '17, IEEE Computer Society, Washington, DC, USA, 2017, pp. 172–179. doi:10.1109/SCC.2017.29.

[17] M. Stumptner, M. Schrefl, Behavior consistent inheritance in UML, in: 9th International Conference on Conceptual Modeling (ER), Springer, 2000, pp. 527–542. doi:10.1007/3-540-45393-8_38.

[18] H. Jin, K. Ravi, F. Somenzi, Fate and free will in error traces, International Journal on Software Tools for Technology Transfer 6 (2) (2004) 102–116. doi:10.1007/s10009-004-0146-9.

[19] T. Ball, M. Naik, S. K. Rajamani, From symptom to cause: Localizing errors in counterexample traces, in: 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, 2003, pp. 97–105. doi:10.1145/604131.604140.

[20] R. Eshuis, R. Wieringa, Verification support for workflow design with UML activity graphs, in: 24th International Conference on Software Engineering (ICSE), ACM, 2002, pp. 166–176. doi:10.1145/581339.581362.

[21] R. Eshuis, Symbolic model checking of UML activity diagrams, ACM Trans. Softw. Eng. Methodol. 15 (1) (2006) 1–38. doi:10.1145/1125808.1125809.

[22] V. S. W. Lam, A formalism for reasoning about UML activity diagrams, Nordic J. of Computing 14 (1) (2007) 43–64.

[23] V. S. W. Lam, Theory for classifying equivalences of unified modeling language activity diagrams., IET Software 2 (5) (2008) 391–403.

[24] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, SFCS '77, IEEE Computer Society, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[25] A. Cimatti, E. M. Clarke, F. Giunchiglia, M. Roveri, NuSMV: A new symbolic model verifier, in: 11th Int'l Conf. on Computer Aided Verification (CAV), Springer-Verlag, 1999, pp. 495–499. doi:10.1007/3-540-48683-6_44.

[26] H. Störrle, On the impact of layout quality to understanding UML diagrams: Size matters, in: 17th International Conference on Model-Driven Engineering Languages and Systems (MoDELS), Springer, 2014, pp. 518–534. doi:10.1007/978-3-319-11653-2_32.

[27] F. UL Muram, H. Tran, U. Zdun, Automated mapping of uml activity diagrams to formal specifications for supporting containment checking, in: 11th Int'l Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA), Open Publishing Association, 2014, pp. 93–107. doi:10.4204/EPTCS.147.7.

[28] F. UL Muram, H. Tran, U. Zdun, Counterexample analysis for supporting containment checking of business process models, in: Business Process Management Workshops - BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 - September 3, 2015, Revised Papers, 2015, pp. 515–528. doi:10.1007/978-3-319-42887-1_41.

[29] R. van der Straeten, T. Mens, J. Simmonds, V. Jonckers, Using description logic to maintain consistency between UML models, in: 6th Int'l Conf. on The Unified Modeling Language, Modeling Languages and Applications, Springer, 2003, pp. 326–340. doi:10.1007/978-3-540-45221-8_28.

[30] B. Graaf, A. van Deursen, Model-driven consistency checking of behavioural specifications, in: Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES'07), IEEE, 2007, pp. 115–126. doi:10.1109/MOMPES.2007.12.

[31] N. Amálio, S. Stepney, F. Polack, Formal proof from uml models, in: Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 418–433. doi:10.1007/978-3-540-30482-1_35.

[32] M. P. Heimdahl, Y. Choi, M. W. Whalen, Deviation analysis: A new use of model checking, Automated Software Engg. 12 (3) (2005) 321–347. doi:10.1007/s10515-005-2642-x.

[33] R. van der Straeten, Inconsistency management in model-driven engineering an approach using description logics, Doctoral dissertation, Vrije Universiteit Brussel (2005).

[34] M. Becker, R. Laue, A comparative survey of business process similarity measures, Computers in Industry 63 (2) (2012) 148–167. doi:10.1016/j.compind.2011.11.003.

[35] R. Dijkman, M. Dumas, L. García-Bañuelos, Graph matching algorithms for business process model similarity search, in: 7th International Conference on Business Process Management (BPM), Springer, Ulm, Germany, 2009, pp. 48–63. doi:10.1007/978-3-642-03848-8_5.

[36] R. Dijkman, M. Dumas, B. van Dongen, R. Käärik, J. Mendling, Similarity of business process models: Metrics and evaluation, Information Systems 36 (2) (2011) 498–516. doi:10.1016/j.is.2010.09.006.

[37] W. M. P. van der Aalst, A. K. A. de Medeiros, A. J. M. M. Weijters, Process equivalence: Comparing two process models based on observed behavior, in: 4th International Conference on Business Process Management (BPM), Springer, 2006, pp. 129–144. doi:10.1007/11841760_10.

[38] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, E. Verbeek, Conformance checking of service behavior, ACM Transactions on Internet Technology 8 (3) (2008) 1–30. doi:10.1145/1361186.1361189.

[39] J. Bae, L. Liu, J. Caverlee, L.-j. Zhang, H. Bae, Development of distance measures for process mining, discovery and integration, International Journal of Web Services Research 4 (4) (2007) 1–17. doi:10.4018/jwsr.2007100101.

[40] W. M. P. van der Aalst, Inheritance of dynamic behaviour in UML, in: 2nd International Workshop on Modelling of Objects, Components and Agents (MOCA), 2002, pp. 105–120.

[41] G. Engels, J. M. Küuster, L. Groenwegen, Consistent interaction of software components, Journal of Integrated Design and Process Science 6 (4) (2002) 2–22.

[42] A. Egyed, Automated abstraction of class diagrams, ACM Trans. Softw. Eng. Methodol. 11 (4) (2002) 449–491. doi:10.1145/606612.606616.

[43] P. Arcaini, A. Gargantini, E. Riccobene, Smt-based automatic proof of ASM model refinement, in: Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings, 2016, pp. 253–269. doi:10.1007/978-3-319-41591-8_17.

[44] S. Krings, M. Leuschel, Proof assisted symbolic model checking for B and event-b, in: Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings, Springer International Publishing, Cham, 2016, pp. 135–150. doi:10.1007/978-3-319-33600-8_8.

[45] J. Koehler, G. Tirenni, S. Kumaran, From business process model to consistent implementation: A case for formal verification methods, in: 6th Int'l Enterprise Distributed Object Computing Conf. (EDOC'02), IEEE Computer Society, 2002, pp. 96–. doi:10.1109/EDOC.2002.1137700.

[46] G. Engels, B. Güldali, C. Soltenborn, H. Wehrheim, Assuring consistency of business process models and web services using visual contracts, in: A. Schürr, M. Nagl, A. Zündorf (Eds.), Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers, Springer-Verlag, 2008, pp. 17–31. doi:10.1007/978-3-540-89020-1_2.

[47] A. Martens, Consistency between executable and abstract processes, in: IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE), IEEE Computer Society, 2005, pp. 60–67. doi:10.1109/EEE.2005.53.

[48] OASIS, Web Services Business Process Execution Language (WSBPEL), https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, (Last accessed: May 20, 2018) (2007).

[49] A. Förster, G. Engels, T. Schattkowsky, R. Van Der Straeten, Verification of Business Process Quality Constraints based on Visual Process Patterns, in: First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE), IEEE, 2007, pp. 197–206. doi:10.1109/TASE.2007.56.

[50] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, P. v. d. Stappen, Model checking for managers, in: Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings, Springer-Verlag, 1999, pp. 92–107. doi:10.1007/3-540-48234-2_7.

[51] A. Wasylkowski, A. Zeller, Mining temporal specifications from object usage, in: IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE Computer Society, 2009, pp. 295–306. doi:10.1109/ASE.2009.30.

[52] K. Y. Rozier, Survey: Linear temporal logic symbolic model checking, Comput. Sci. Rev. 5 (2) (2011) 163–203. doi:10.1016/j.cosrev.2010.06.002.

[53] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic model checking: 10ˆ20 states and beyond, Inf. Comput. 98 (2) (1992) 142–170. doi:10.1016/0890-5401(92)90017-A.

[54] Y. Dong, Z. Shensheng, Using pi-calculus to formalize uml activity diagram for business process modeling, in: 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, IEEE Computer Society, 2003, pp. 47–54. doi:10.1109/ECBS.2003.1194782.

[55] N. Guelfi, A. Mammar, A formal semantics of timed activity diagrams and its promela translation, in: 12th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, IEEE Computer Society, 2005, pp. 283–290. doi:10.1109/APSEC.2005.7.

[56] D. Harel, A. Naamad, The statemate semantics of statecharts, ACM Trans. Softw. Eng. Methodol. 5 (4) (1996) 293–333. doi:10.1145/235321.235322.

[57] H. Eshuis, Semantics and verification of uml activity diagrams for workflow modelling, Ph.D. thesis, Univ. of Twente, CTIT Ph.D.-thesis series No. 02-44 (November 2002).

[58] B. Lerner, S. Christov, L. Osterweil, R. Bendraou, U. Kannengiesser, A. Wise, Exception handling patterns for process modeling, Software

Engineering, IEEE Transactions on 36 (2) (2010) 162–183. doi:10.1109/TSE.2010.1.

[59] E. M. Clarke, Jr., O. Grumberg, D. A. Peled, Model Checking, MIT Press, 1999.

[60] D. M. Gabbay, The declarative past and imperative future: Executable temporal logic for interactive systems, in: Temporal Logic in Specification, Springer-Verlag, London, UK, UK, 1987, pp. 409–448.

[61] E. M. Clarke, K. L. McMillan, S. V. A. Campos, V. Hartonas-Garmhausen, Symbolic model checking, in: 8th Int'l Conf. on Computer Aided Verification (CAV), Springer, 1996, pp. 419–427. doi:10.1007/3-540-61474-5_93.

[62] R. Cavada, A. Cimatti, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev, Nusmv 2.5 user manual, http://http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf, (Last accessed: June 2, 2018) (2005).

[63] R. Milner, Communication and Concurrency, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[64] T. Murata, Petri Nets: Properties, Analysis and Applications, Proceedings of the IEEE 77 (4) (1989) 541–580. doi:10.1109/5.24143.

[65] F. Pelletier, Ternary Exclusive Or, Logic Journal of the Igpl 16 (1) (2008) 75–83. doi:10.1093/jigpal/jzm027.

[66] H. Tran, U. Zdun, T. Holmes, E. Oberortner, E. Mulo, S. Dustdar, Compliance in service-oriented architectures: A model-driven and view-based approach, Information & Software Technology 54 (6) (2012) 531–552, special Section: Engineering Complex Software Systems through Multi-Agent Systems and Simulation. doi:10.1016/j.infsof.2012.01.001.

[67] H. Tran, T. Holmes, U. Zdun, S. Dustdar, Using model-driven views and trace links to relate requirements and architecture: A case study, in: P. Avgeriou, J. Grundy, J. G. Hall, P. Lago, I. Mistrík (Eds.), Relating Software Requirements and Architectures, Springer, 2011, pp. 233–255. doi:10.1007/978-3-642-21001-3_14.

[68] E. M. Clarke, J. M. Wing, Formal methods: State of the art and future directions, ACM Comput. Surv. 28 (4) (1996) 626–643. doi:10.1145/242223.242257.

[69] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Property specification patterns for finite-state verification, in: Second Workshop on Formal Methods in Software Practice (FMSP), ACM, 1998, pp. 7–15. doi:10.1145/298595.298598.

[70] H. Tran, U. Zdun, S. Dustdar, Name-based view integration for enhancing the reusability in process-driven soas, in: Business Process Management Workshops - BPM 2010 International Workshops and Education Track, Revised Selected Papers, Springer, 2010, pp. 338–349. doi:10.1007/978-3-642-20511-8_32.