Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking

Faiz UL Muram Huy Tran Uwe Zdun
Research Group Software Architecture
University of Vienna, Austria.
faiz.ulmuram|huy.tran|uwe.zdun@univie.ac.at

Business analysts and domain experts are often sketching the behaviors of a software system using high-level models that are technology- and platform-independent. The developers will refine and enrich these high-level models with technical details. As a consequence, the refined models can deviate from the original models over time, especially when the two kinds of models evolve independently. In this context, we focus on behavior models; that is, we aim to ensure that the refined, low-level behavior models conform to the corresponding high-level behavior models. Based on existing formal verification techniques, we propose containment checking as a means to assess whether the system's behaviors described by the low-level models satisfy what has been specified in the high-level counterparts. One of the major obstacles is how to lessen the burden of creating formal specifications of the behavior models as well as consistency constraints, which is a tedious and error-prone task when done manually. Our approach presented in this paper aims at alleviating the aforementioned challenges by considering the behavior models as verification inputs and devising automated mappings of behavior models onto formal properties and descriptions that can be directly used by model checkers. We discuss various challenges in our approach and show the applicability of our approach in illustrative scenarios.

1 Introduction

Behavior models are used in many areas of software engineering. Examples of behavior models are UML activity diagrams, sequence diagrams, and state charts [12] as well as BPMN business processes[1], BPEL business processes[2], Event-driven Process Chains (EPCs) [19], to name but a few. Many models are created as "high-level" models. That is, they are mainly used to convey the core concepts or principles of the reality they represent in an abstract and/or concise way. They are used for tasks such as defining core concepts and principles of a domain, enabling stakeholders to discuss them for instance in the course of a system design, or creating a common terminology. Also, high-level models are used as abstractions. For example, in design and architecture patterns high-level models are used as abstract representations of a best practice which need to be concretized in real systems. On the other hand, technical or "low-level" models are often created with purposes such as providing a precise specification of the source code, executing the model (e.g., in a process engine, interpreter, or virtual machine), or generating executable code directly from the model, e.g., in model-driven development (MDD).

[1] http://www.omg.org/spec/BPMN/2.0
[2] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

Submitted to:
FESCA @ ETAPS 2014

Unfortunately, the high-level models and their low-level counterpart are often drifting apart over time if, for example, contradictory specifications are defined. Therefore, it is important to ensure that the behaviors represented by the low-level models conform to what has been specified in the high-level models. In the literature, several techniques have been exploited to check the consistency and containment of different types of behavioral models [15]. Essentially, these approaches represent behavioral models in terms of formal specifications and/or properties and perform model checking to verify whether the formal specifications and properties are consistent. Most of these approaches assume that the formal specifications and properties are readily available or can be easily created beforehand. In reality, however, achieving such formal specifications is a tedious and challenging task because it often requires considerable knowledge on the leveraged formal techniques and it is often accomplished in a laborious, manual manner.

In this paper, we propose a novel approach to alleviating the aforementioned problems. First, unlike most of existing approaches, we consider the high-level and low-level behavior models as inputs for checking containment. Thus, extra effort in creating consistency constraints that are widely used in the existing approaches can be reduced. Second, we devise an automated mapping of behavior models to formal specifications that can be used directly by model checkers for containment verification purpose. In particular, we derive primitive patterns for representing fundamental behavioral constructs such as sequences, parallel structures, and branches. These patterns are associated with formal description structures, for instance, linear temporal logic (LTL) [16, 18] and the symbolic model language SMV [3, 4]. Based on model-driven transformation techniques, our approach enables automated generation of SMV descriptions and LTL formulas from given input behavioral models such as (in our case) UML activity diagrams [12]. Using the NuSMV model checker[3], we can assess whether a low-level technical model or implementation is consistent with the specification provided by a high-level, behavior model. In our work, we target UML activity diagrams because they are widely used in both industry and academia for representing software systems behavior. Linear temporal logic and SMV are used as the underlying formalism because, on the one hand, their expressiveness is suitable for model consistency and containment checking. On the other hand, we can leverage existing powerful tools for LTL and SMV model checking. Please note that our approach is also applicable for other behavioral models, such as the ones mentioned above, and can also be realized with different formal techniques with reasonable extra effort.

The paper is structured as follows. In Section 2, we review the related approaches on supporting behavioral consistency checking in general and containment checking in particular. Our approach for automatically translating software behavior models onto formal properties and specifications is presented in detail in Section 3 along with illustrative examples. Afterwards, we discuss various challenges in supporting the aforementioned automated mappings in Section 4. Finally, we summarize the main contributions and discuss the planned future work in Section 5.

2 Related Work

A considerable amount of consistency checking approaches have been proposed so far. Finkelstein et al. present an approach using temporal logic for checking the consistency of different viewpoints [10]. Similar approaches targeting different kinds of models or model checking techniques exist [15]. While some of those focus on behavior models, the major difference to our work is that those

[3] `http://nusmv.fbk.eu`

other approaches mainly consider the consistency between different kinds of models (or models and other representations of the same reality such as the implementations or requirements) but not the consistency of the same model at different abstraction levels. That is, 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. According to the methodology to handle consistency problems proposed by Engels et al. [8], we address the "vertical consistency"—in contrast to "horizontal consistency" which checks consistency between models.

Additionally, most of the existing approaches require the specification of a consistency condition for each consistency problem addressed. The goal of our approach in contrast is to use a high-level model, which is automatically translated to formal descriptions, as the specification of the consistency problem and check the low-level model whether it contains the high-level specifications.

There are only a few approaches for containment checking that use high-level models as inputs but they merely focus on structural models. Egyed introduces an approach based on structural transformation rules and transitive reasoning to check whether an UML class diagram conforms to another more abstract class diagram [6]. Unfortunately, this approach alone does not suffice for the behavior models addressed in our work because we must assume that behavior models contain control structures, such as sequences, links, parallel gateways, and so on, which cannot be matched by structure only.

Many other consistency checking approaches focus on different types of consistency but containment. Van Der Straeten et al. check the consistency of different UML models using a description logic based approach [20]. This approach does not focus on consistency of high-level and low-level models, but rather on model-model, model-instance, and instance-instance conflicts. Ehrig and Tsiolakis use attributed graphs and graphical constraints to check the consistency of UML class and sequence diagrams [21]. In particular, the existence, correct multiplicity, and valid scoping of model elements are checked. Graaf and van Deursen introduce a model-driven consistency checking approach for checking the consistency of various behavior models among each other [11]. The approach first normalizes the input models, then performs an automated model transformation to a state machine, and then compares the different state machines to detect inconsistencies.

There are a number of efforts on translating manually described constraints to formal representations. For instance, Czarnecki and Pietroszek check the well-formedness of feature diagrams using OCL constraints (i.e. structural models) [5]. Engels et al. check contracts between Web services and business processes (i.e., consistency between behavior and structural models) [7]. Campbell et al. check for and visualize errors in UML diagrams [2]. Köhler et al. introduce an approach to verifying a business process model and its implementation but merely concentrate on checking the termination property [13]. Eshuis considered using NuSMV for checking data integrity constraints between an activity diagram and a set of class diagrams [9].

We note that, in most of the aforementioned approaches, the availability of the input formal specifications is often taken for granted. Unfortunately, achieving the formal specifications as inputs for model checking techniques is challenging and error-prone for the majority of developers as reasonable knowledge of formal languages and verification techniques is always necessary. Our approach aims at filling this gap by introducing primitive behavior patterns grounded on formal expressions that can support the automated mapping of the high-level and low-level behavior models to formal specifications.

3 Approach

In this section, we firstly present an overview of our approach focusing on the automated mapping of behavior models into formal descriptions for containment checking. Afterwards, we elaborate on our proposed techniques for mapping high-level behavior models onto formal property specifications (i.e. LTL formulas in our case) and the low-level counterparts into formal SMV specifications.

3.1 Approach overview

Our containment checking approach addresses the consistency problems existing between high-level and low-level behavior models. The low-level counterparts are often resulting from various steps of refining and enriching the high-level models. In most of the cases, models evolve, often independently, over time. For instance, high-level models are changed according to new requirements, and low-level models are changed as the implementation is modified. As a consequence, the evolved models may include inconsistencies, such as the order of activities swaps during refinement of one of the models, some new activities are added in the high-level model but are not present in the low-level model, or some low-level activities are deleted without updating the high-level model.

The main goal of checking containment is to verify whether the behavior described by a low-level model still satisfies the behavior specified by a corresponding high-level model. In other words, *containment checking will assess if the execution traces produced by the low-level model contain those produced by the high-level counterpart.* Assuming that the low-level behavior model is represented by a formal specification, we could achieve containment checking by checking that the high-level model's desired properties are satisfied by this formal specification. As a result, a model checker can be used to answer whether the formal specification satisfies these properties, i.e., the low-level behavior model conforms to the corresponding high-level model. Unfortunately, achieving formal representations and properties of behavior models under consideration is a challenging and error-prone task because it requires deep knowledge of the formal verification techniques.

Therefore, we present an approach to address this challenge by supporting automated translation of behavior models into formal specifications and properties that can be used by model checkers for containment checking. The main focus of our approach is represented by the grey box shown in Figure 1. In particular, we introduce primitives for representing fundamental behavioral constructs, such as sequences, parallel structures, and branches, in terms of LTL formulas. These primitives will be used to transform high-level UML activity diagrams into LTL properties. The low-level UML activity diagrams under consideration will be transformed to symbolic formal specifications. We note that these transformations are represented as solid arrows in Figure 1 as they will be performed automatically.

Finally, containment checking is performed on these models by using the NuSMV model checker, which supports symbolic model verification [4]. By analyzing the verification results reported by the NuSMV model checker, one can assess whether the input behavior models are consistent as well as reason about their inconsistency (if any exists) through the generated counterexamples. Verification and reasoning with the NuSMV model checker is, however, beyond the scope of this paper and will be part of our follow-on endeavor.

The aforementioned concepts have been realized in our prototypical implementation. In

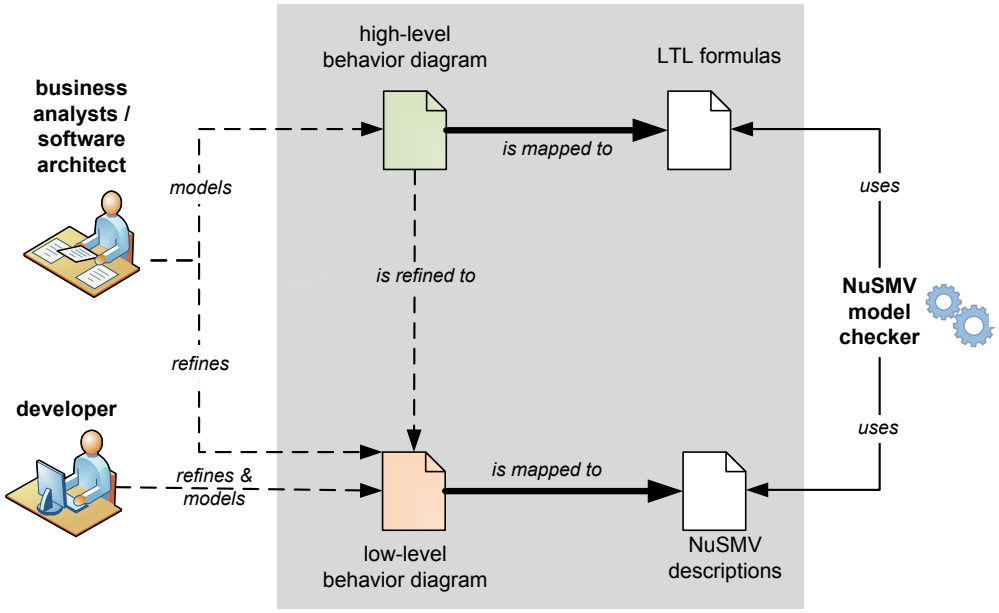


Figure 1: Overview of the approach

our implementation, we exploit model-driven development techniques to support automated transformations of UML activity diagrams. In particular, we leverage Eclipse Xtend framework[4], which provides powerful and expressive languages and techniques for defining and executing transformation rules. As a result, our implementation can be easily integrated in the Eclipse development environment.

3.2 Transformation of High-level UML Activity Diagrams into LTL Formulas

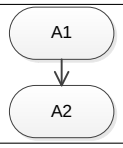
The first aspect of our approach is to map a set of fundamental behavioral constructs of UML activity diagrams, such as sequences, parallel structures, and branches, into LTL-based properties. LTL is an expressive formalism that is widely used in formal verification tools. Formulas in LTL are usually constructed from boolean predicates and logical operators such as *not* (!), *or* (|), *and* (&), *xor*, and *implication* ($\rightarrow$). LTL also supports the specification of future with temporal operators such as '*always*' (G), '*next*' (X), '*eventually*' (F), and so on. For more details on LTL formulas as well as their semantics, please refer to [18].

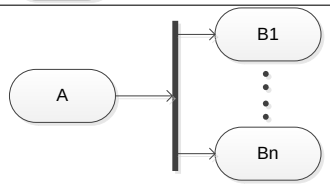
According to the OMG UML 2.4 specification [12], UML activity diagrams provide several constructs for describing the behavior of software systems. Without loss of generality, we consider a set of fundamental elements and control structures such as activities, actions, sequences, parallel structures (fork and join), branching (decision and merge) in the scope of this paper. Other constructs are also applicable using our approach in similar manner with little extra effort. The *xor* operator is often used in the LTL-based primitive for representing decision [1].

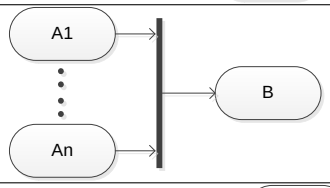
In Table 1, we present these constructs of UML activity diagrams and their informal description extracted from the UML specification [12]. The right-hand side column contains the corresponding LTL-based primitive patterns for formally representing these constructs. Using these patterns,

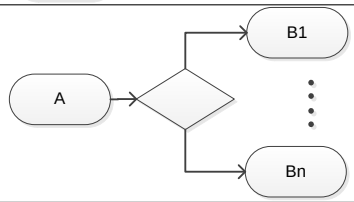
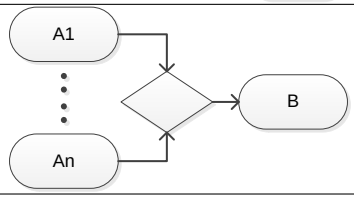
[4] `http://www.eclipse.org/xtend/documentation/2.4.3/Documentation.pdf`

Table 1: Representation of UML activity diagram's elements using LTL primitives

Description	Modeling Notation	LTL Primitive
Sequence: A set of actions executed in sequential order. E.g., the action A1 will be performed before A2.		`G (A1 -> F A2)`

Fork Node: The execution of a fork node leads to the parallel execution of subsequent actions (B1...Bn).		`G (A -> (F B1`
		`& F B2`
		`& ...`
		`& F Bn))`

Join Node: The execution of two or more parallel actions leads to the execution of Join Node.		`(G (A1`
		`& A2`
		`& ...`
		`& An) -> F B)`

Decision Node: The execution of a Decision Node eventually followed by the execution of one and only one action among the available set of actions based on the decision guard.		`G (A -> (F B1 xor F B2` `xor ... xor F Bn))`
Merge Node: At least one action among a set of alternative execution of actions will lead to the execution of Merge Node.		`G (A1 | A2 | ... | An ->` `F B)`

we traverse through the input UML activity diagram (i.e. the high-level model) and translate its elements into corresponding LTL formulas.

We use a simplified order processing system example to demonstrate the transformation of UML activity diagrams to LTL formulas. The high-level model of the order processing system is shown in Figure 2 and it includes sequential actions, a decision node, a fork node, and a join node. Using the primitives shown in Table 1, we are able to generate the corresponding LTL formulas in Listing 1 out of the aforementioned UML activity diagram.

```
1 LTLSPEC G (InitialNode1 -> F VerifyCreditCard)
2 LTLSPEC G (VerifyCreditCard -> F ReplyCreditCardNotOK xor F CreateOrderBusinessObject)
3 LTLSPEC G (ReplyCreditCardNotOK -> F ActivityFinalNode1)
4 LTLSPEC G (CreateOrderBusinessObject -> F ShipOrder & F ChargeOrder)
5 LTLSPEC (G (ShipOrder & ChargeOrder) -> F ReplyOrderStatus)
6 LTLSPEC G (ReplyOrderStatus -> F ActivityFinalNode2)
```

Listing 1: The formal property specifications automatically generated from the high-level UML activity diagram

The automated transformation of high-level UML activity diagrams into LTL formulas has been realized using Eclipse Xtend. An excerpt of the Xtend template expression for describing

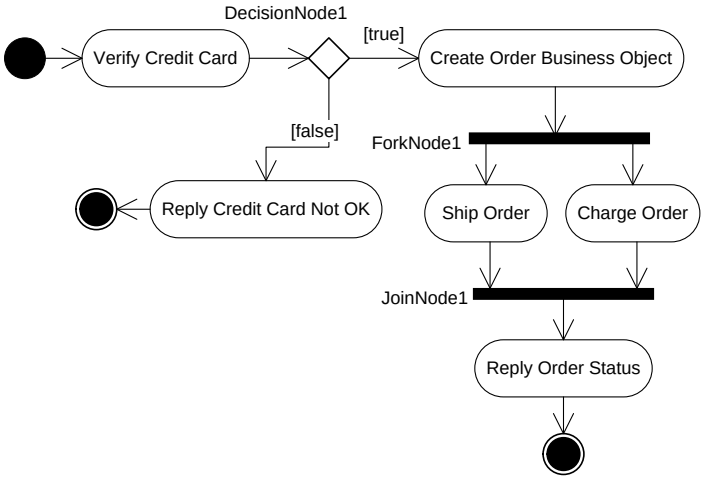


Figure 2: High-level UML activity diagram

the transformation of basic constructs such as sequence and fork node into corresponding LTL formulas is shown in Listing 2. Xtend templates expressions are multi-line strings, that are surrounded by triple single quotes (”’). These strings are further enhanced with expressions embraced in a pair of guillemets (i.e., “«” and “»”). Such expressions will be bound and evaluated according to the elements of the input UML activity diagram.

Listing 2: Transformation of UML structures into LTL formulas

```
/* For the generation of sequential actions */
«IF (!convert(a).nullOrEmpty)»LTLSPEC G («a.name» -> F «convert(a)»);«ENDIF»
/* Template expression for the generation of a fork node */
«IF (!convert(a).nullOrEmpty)»LTLSPEC G («a.incomings.iterator().next().normalize» -> F «convert(a)»);
«ENDIF»
/* Create a Fork's guard condition */
 a.outgoings.forEach [ edge, i |
      val target = edge.target
      if (edge != null && target != null) {
           if (i > 0)
             guard.append( " & F ")
           guard.append(target.name)
      }
  ]
```

The first template presents the mapping of actions into LTL formulas that are executed in the sequence. The second template defines generation of LTL formulas for the fork node. To this, first we get the incoming flow that triggers the fork node and map them into LTL rule. Afterward, the subsequent action(s) that are executed by the fork node are mapped into LTL rules. If there are more than one outgoing actions then ‘`& F`’ is attached with every subsequent actions. These LTL formulas are generated according to the primitives presented in Table 1.

3.3 Transformation of Low-level UML Activity Diagrams into SMV specifications

The second part of our approach automates the translation of low-level UML activity diagrams into formal SMV specifications. These specifications then can be used by the NuSMV model checker to verify against the LTL rules generated from the high-level counterparts to see whether

the containment relationship between these diagrams is satisfied. The transformation templates for the UML activity diagram constructs considered in this paper are shown in Table 2.

The transformation of a low-level UML activity diagram into SMV specification works as follows. Essentially, every node of a UML activity diagram except a decision node will be represented by a corresponding symbolic variable declared in the '`VAR`' section. A decision node, as specified in UML 2.4 specification [12], will trigger the execution of one of its branching nodes according to their guarded conditions. Therefore, we map a decision node into a scalar variable. The value of the scalar variable is defined based on the values of the branch guarded conditions plus the constant 'undetermined' used as initial state.

After having nodes represented as symbolic variables, we need to map the control flow of the UML activity diagram into the corresponding state transition rules for each variable. The general description of a variable's state transitions is a combination of the statement `init()`–defining the initial state–and `next()`–defining the next state. As a result, given a certain node n of a UML activity diagram represented by a NuSMV symbolic variable a, we need to define a set of statements as follows: "`init(a) := initial_value; next(a) := next_value;`".

The `next_value` can be defined by a concrete value straightforwardly or through an exclusive choice of various possible values with respect to some constraints or previous states. In the later case, we can use the NuSMV construct "`case...esac`" to specify such exclusive choices. Thus, the most plausible form of the state transitions is to (1) initialize the variable's state as `FALSE`; (2) define a guard condition that trigger the execution of the node according to the UML specification [12] such that the variable's state can become `TRUE`; (3) define another choice to switch the variable's state back to `FALSE` if it was `TRUE` before that. These state transitions can be described in the following NuSMV code.

```
init(a) := initial_value; -- (1)
next(a) := case
  guard_condition : TRUE; -- (2)
  a : FALSE;              -- (3)
esac;
```

The mapping rules shown in Table 2 define the state transition for different types of nodes of a UML activity diagram. The execution of the nodes of an activity diagram is based on token semantics [12] similar to Petri Nets [17]. The `Initial Nodes` are the starting points for a UML activity diagram, therefore, their initial state will be `TRUE`. The initial state of other nodes (except Decision Node) are `FALSE`. Apart from the `Initial Nodes` that have no incoming edges, other nodes will be triggered with respect to their incoming control flows. We note that the UML specification denotes an "implicit join" in case a node has multiple incoming edges. Therefore, the guard for transitioning to the next state for a `Final Node`, `Action`, `Decision Node`, or `Fork Node` is similar to the default semantics of an explicit `Join Node`, i.e., an "*and*"-join of all tokens going through the incoming control flows. We use the operator logical *and* ("`&`") of NuSMV to represent the *and*-join guard. Please note that the non-default "`joinSpec`" of an explicit `Join Node` can also be easily supported by altering the logical condition of the `Join Node` in NuSMV.

A `Merge Node` is a special case, as it brings together multiple alternative flows. Therefore, we use the operator *or* "`|`" of NuSMV to describe the guard condition of a `Merge Node`. The possible states of a symbolic variable representing a node (except Decision Node) are `TRUE` and `FALSE`. In case of a `Decision Node`, the initial state is predefined as `undetermined` and the possible next states is an exclusive choice of its outgoing branches. We use the notion of *non-deterministic assignments* in NuSMV to describe the outcome of a `Decision Node` such that the NuSMV

UML constructs	Transformation Rules

Initial Node *a*	`init(a) := TRUE;`
	`next(a) := case`
	`  a : FALSE;`
	`  TRUE : a;`
	`esac;`

Final Node, Action, Fork Node, Join Node *a*	`init(a) := FALSE;`
	`next(a) := case`
	`  incoming_1 & incoming_2 & ... & incoming_n : TRUE;`
	`  a : FALSE;`
	`esac;`

Decision Node *a*	`init(a) := undetermined;`
	`next(a) := case`
	`  incoming_1 & incoming_2 & ... & incoming_n$ : {outgoing_1, outgoing_2, ..., outgoing_n};`
	`  a != undetermined : undetermined;`
	`esac;`

Merge Node	`init(a) := FALSE;`
	`next(a) := case`
	`  incoming_1 | incoming_2 | ... | incoming_n : TRUE;`
	`  a : FALSE;`
	`esac;`

Table 2: Transformation of UML structures into SMV descriptions

model checker can explore all possible outcomes for verification. We have implemented these rules using the Eclispe Xtend language and exploit the code generation template of Xtend to automatically produce NuSMV descriptions out of a low-level UML activity diagram.

To illustrate the transformation of low-level UML activity diagram into NuSMV, we use one of the refined versions of the order processing system mentioned above. The low-level model of the order processing system is shown in Figure 3. In this diagram, technical details are added that are required for an implementation of the order processing system, but have been omitted by the domain experts.

The transformation rules explained above can be used to automatically produce a corresponding NuSMV description. First, the nodes of the activity diagram (except decision nodes) are represented by a boolean symbolic variable as shown in Listing 3. Because a decision node can assign the output token to one of the outgoing branches, we use a scalar variable to represent a decision node. The range of values of the scalar variable correspond to the guards of the outgoing branches of the decision node. Please note that we have given names for all structural nodes in Figure 3 to make the example mapping to the NuSMV description below easily understandable. This is not necessary in our approach, our code generator could also automatically assign names.

```
MODULE
VAR
  InitialNode1 : boolean;
  ReceiveNewOrder : boolean;
  VerifyCreditCard : boolean;
  DecisionNode1 : {undetermined, guard_DecisionNode1_ReplyCreditCardNotOK,
      guard_DecisionNode1_DecisionNode2};
  DecisionNode2 : {undetermined, guard_DecisionNode2_ConfirmOrderCancelation,
      guard_DecisionNode2_CreateOrderBusinessObject};
  -- partially omitted
```

Listing 3: NuSMV symbolic variables for representing UML activity diagram nodes

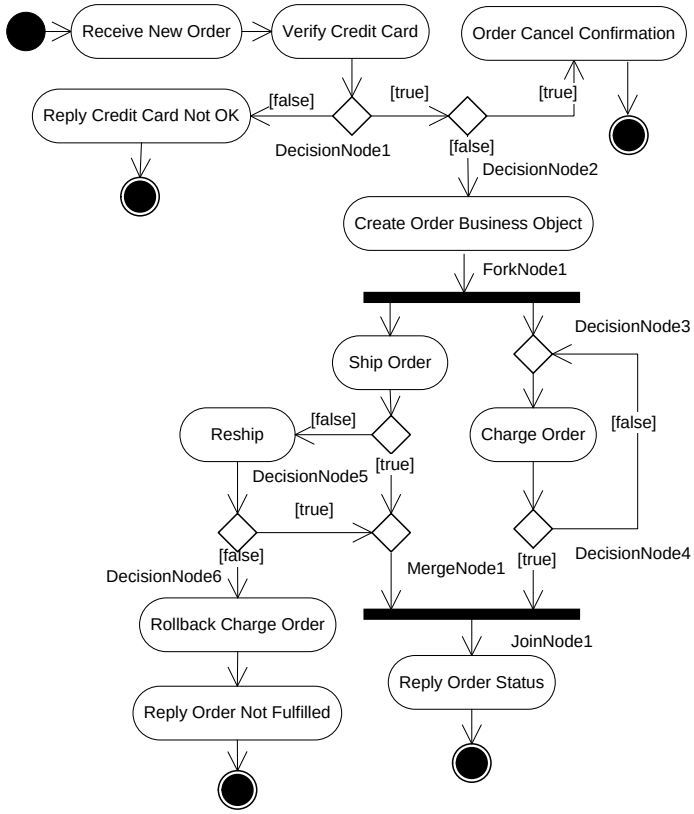


Figure 3: Low-level UML activity diagram for the order processing system

We show in Listing 4 an excerpt of the corresponding state transitions of the aforementioned symbolic variables according to the rules in Table 2. Some repetitive excerpts, for instance, produced from the similar types of nodes, have been omitted due to space limitation.

```
ASSIGN
init(InitialNode1) := TRUE;
next(InitialNode1) := case
       InitialNode1 : FALSE;
    esac;
init(ReceiveNewOrder) := FALSE;
next(ReceiveNewOrder) := case
       InitialNode1 : TRUE;
       ReceiveNewOrder : FALSE;
    esac;
...
init(DecisionNode1) := undetermined;
next(DecisionNode1) := case
       VerifyCreditCard : {guard_DecisionNode1_ReplyCreditCardNotOK, guard_DecisionNode1_DecisionNode2};
       DecisionNode1 != undetermined : undetermined;
    esac;
init(ReplyCreditCardNotOK) := FALSE;
next(ReplyCreditCardNotOK) := case
       (DecisionNode1 = guard_DecisionNode1_ReplyCreditCardNotOK) : TRUE;
       ReplyCreditCardNotOK : FALSE;
    esac;
init(DecisionNode2) := undetermined;
next(DecisionNode2) := case
       (DecisionNode1 = guard_DecisionNode1_DecisionNode2) : {
```

```
            guard_DecisionNode2_ConfirmOrderCancelation, guard_DecisionNode2_CreateOrderBusinessObject};
        DecisionNode2 != undetermined : undetermined;
    esac;
init(ActivityFinalNode1) := FALSE;
next(ActivityFinalNode1) := case
        ReplyCreditCardNotOK : TRUE;
        ActivityFinalNode1 : FALSE;
    esac;
init(ConfirmOrderCancelation) := FALSE;
next(ConfirmOrderCancelation) := case
        (DecisionNode2 = guard_DecisionNode2_ConfirmOrderCancelation) : TRUE;
        ConfirmOrderCancelation : FALSE;
    esac;
init(CreateOrderBusinessObject) := FALSE;
next(CreateOrderBusinessObject) := case
        (DecisionNode2 = guard_DecisionNode2_CreateOrderBusinessObject) : TRUE;
        CreateOrderBusinessObject : FALSE;
    esac;
-- generated code is partially omitted
init(ForkNode1) := FALSE;
next(ForkNode1) := case
        CreateOrderBusinessObject : TRUE;
        ForkNode1 : FALSE;
    esac;
init(ShipOrder) := FALSE;
next(ShipOrder) := case
        ForkNode1 : TRUE;
        ShipOrder : FALSE;
    esac;
init(MergeNode2) := FALSE;
next(MergeNode2) := case
        (DecisionNode4 = guard_DecisionNode4_MergeNode2) | ForkNode1 : TRUE;
        MergeNode2 : FALSE;
    esac;
init(DecisionNode5) := undetermined;
next(DecisionNode5) := case
        ShipOrder : {guard_DecisionNode5_MergeNode1, guard_DecisionNode5_Reship};
        DecisionNode5 != undetermined : undetermined;
    esac;
-- generated code is partially omitted
```

Listing 4: NuSMV state transitions for the low-level UML activity diagram

3.4 Illustration of Containment Checking Using NuSMV

In this section, we illustrate how the LTL properties and NuSMV descriptions generated in the previous steps can be used for containment checking. We use the NuSMV model checker to perform containment checking based on the formal SMV specifications illustrated in Figure 3 and the LTL rules generated from the high-level counterparts presented in Listing 1. The containment checking result is shown in Listing 5. By looking at the result, we see that the generated NuSMV description satisfies all LTL properties except "`LTLSPEC G (VerifyCreditCard -> F ReplyCreditCardNotOK xor F CreateOrderBusinessObject)`". This unsatisfied condition is due to the occurrence of the `Decision Node 2` that leads to two possibilities outcomes, which are `CreateOrderBusinessObject` and `OrderCancelConfirmation`. In other words, the verification result indicates that the descriptions of the low-level order processing model do not conform the LTL formulas generated from the high-level counterparts. According to the specification, `ReplyCreditCardNotOK` should becomes true in the future. But in the counter example, possible loop is presented and marked with the "Loop starts here" line. The line indicates that the

`ReplyCreditCardNotOK` never becomes true in the loop, so the behavior can occur repeatedly. Hence, this LTL specification is false.

Listing 5: NuSMV containment checking result along with a counterexample

```
$ NuSMV LowlevelOrderProcessingNotSatisfied.smv
-- specification  G (InitialNode1 ->  F VerifyCreditCard)  is true
-- specification  G (VerifyCreditCard -> ( F ReplyCreditCardNotOK xor  F
    CreateOrderBusinessObject))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  InitialNode1 = TRUE
  ReceiveNewOrder = FALSE
  VerifyCreditCard = FALSE
  DecisionNode1 = undetermined
  ReplyCreditCardNotOK = FALSE
  DecisionNode2 = undetermined
  ActivityFinalNode1 = FALSE
  ConfirmOrderCancelation = FALSE
  ...
-> State: 1.2 <-
  InitialNode1 = FALSE
  ReceiveNewOrder = TRUE
-> State: 1.3 <-
  ReceiveNewOrder = FALSE
  VerifyCreditCard = TRUE
-> State: 1.4 <-
  VerifyCreditCard = FALSE
  DecisionNode1 = guard_DecisionNode1_DecisionNode2
-> State: 1.5 <-
  DecisionNode1 = undetermined
  DecisionNode2 = guard_DecisionNode2_ConfirmOrderCancelation
-> State: 1.6 <-
  DecisionNode2 = undetermined
  ConfirmOrderCancelation = TRUE
-> State: 1.7 <-
  ConfirmOrderCancelation = FALSE
  ActivityFinalNode2 = TRUE
-- Loop starts here
-> State: 1.8 <-
  ActivityFinalNode2 = FALSE
-> State: 1.9 <-
-- specification  G (ReplyCreditCardNotOK ->  F ActivityFinalNode1)  is true
...
```

We can see that a corresponding counterexample is also produced by the NuSMV model checker with respect to the unsatisfied properties (see Listing 5). By analyzing the counterexample, we can track down the inconsistency between the two models to know where the containment property has not been satisfied. As our approach presented in this paper mainly focuses on defining containment checking and automatic generating of the formal constraints and specifications as inputs for containment checking, we did not go deeper into the verification process or the automated analysis of the verification result yet (which we plan to do in our future work).

In Figure 4 we illustrate another version of the low-level UML activity diagram of the order processing system that satisfies the containment checking constraints. The checking of the NuSMV description generated out of this model against the LTL formulas from Listing 1 produces no unsatisfied properties, i.e., the NuSMV model checker returns true for all LTL formulas. This result indicates that the low-level behavior model is consistent with its high-level counterparts.

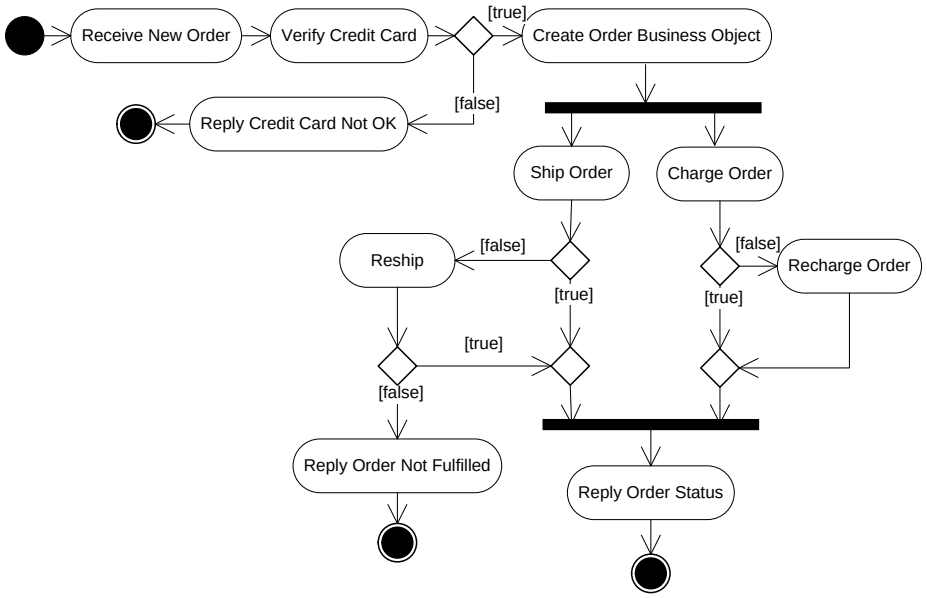


Figure 4: A different version of the low-level UML activity diagram of the order processing system

4 Discussion

This section discusses some issues regarding the automated mappings of behavior models into formal specifications and properties used for containment checking. We have presented the automated mapping of basic constructs of UML activity diagrams like actions, activities, parallel nodes (fork and join), sequences, and branching nodes (decision and merge). At the current level of development, our approach has some limitations, for example, the automated mapping of other constructs of UML activity diagrams such as accept event actions, activity parameter nodes, and exception handling have not been considered yet. But we expect that the automated mapping of these constructs can be achieved in a similar manner with some extra effort. Furthermore, our current work has not incorporated the mapping of loops, data objects and object flows into LTL formulas. The UML 2 specification allows to complement the control flows of a UML activity diagram with data and object flows but does not allow a mix of object flows and control flows. Therefore, it is possible to specify data and object flows separately using the same techniques presented in this paper. Loops might introduce bigger challenges because a loop can produce deterministically or nondeterministically cyclic execution flows that is difficulty to transform to simple LTL constraints. We will investigate other variants of temporal logic such as bounded linear temporal logic [14] to overcome this challenge. These are, however, beyond the scope of this paper and part of our future endeavor. As future work, we also plan to apply our approach on realistic industrial systems. This will also be the basis to further investigate whether our proposed approach is able to support containment checking of software systems in practice.

5 Conclusion

In this paper, we present a novel approach for supporting containment checking of UML activity diagrams. On the one hand, the high-level UML diagrams, often used by business analysts and/or

domain experts, are translated to LTL. On the other hand, the low-level counterparts, often resulting from various steps of refinement and enriching of these high-level models, are mapped onto formal NuSMV descriptions. By considering high- and low-level UML activity diagrams as inputs for containment checking and automatically transforming them into formal properties and descriptions, our approach can help lessening the gap and efforts for creating consistency constraints as many other approaches. The satisfaction of the NuSMV descriptions with respect to the LTL formulas, which can be verified using existing model checking tools, can denote the containment relationship between the high-level and low-level UML activity diagrams.

However, it is difficult to covered whole UML specification in the scope of this paper. Therefore, we have mainly investigated fundamental elements and control structures of UML activity diagrams that are widely used for modeling software system behavior. Nevertheless, the same methods and techniques can also be adapted and applied for other control structures. Therefore, one of our future endeavors is to exploit our approach for various aspects of software behavior modeling such as data objects, object flows, error and exception handling, etc. Another aspect is to consider whether the performance our approach is reasonable in the context of typical software development environments.

Acknowledgment. The research leading to these results has received funding from the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001.

References

[1] Marco Brambilla, Alin Deutsch, Liying Sui & Victor Vianu (2005): *The Role of Visual Tools in a Web Application Design and Verification Framework: A Visual Notation for LTL Formulae.* In David B. Lowe & Martin Gaedke, editors: *ICWE*, *Lecture Notes in Computer Science* 3579, Springer, pp. 557–568. Available at `http://dblp.uni-trier.de/db/conf/icwe/icwe2005.html#BrambillaDSV05`.

[2] Laura A. Campbell, Betty H. C. Cheng, William E. McUmber & Kurt Stirewalt (2002): *Automatically Detecting and Visualising Errors in UML Diagrams. Requir. Eng.* 7(4), pp. 264–287, doi:10.1007/s007660200020.

[3] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia & Marco Roveri (1999): *NUSMV: A New Symbolic Model Verifier.* In: *11th Int'l Conf. on Computer Aided Verification (CAV)*, Springer-Verlag, London, UK, UK, pp. 495–499.

[4] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos & Vassili Hartonas-Garmhausen (1996): *Symbolic Model Checking.* In: *8th Int'l Conf. on Computer Aided Verification (CAV)*, pp. 419–427, doi:10.1007/3-540-61474-5_93.

[5] Krzysztof Czarnecki & Krzysztof Pietroszek (2006): *Verifying feature-based model templates against well-formedness OCL constraints.* In: *5th Int'l Conf. on Generative programming and component engineering (GPCE)*, ACM, New York, NY, USA, pp. 211–220, doi:10.1145/1173706.1173738.

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

[7] Gregor Engels, Baris Güldali, Christian Soltenborn & Heike Wehrheim (2008): *Assuring Consistency of Business Process Models and Web Services Using Visual Contracts.* In Andy Schürr, Manfred Nagl & Albert Zündorf, editors: *Applications of Graph Transformations with Industrial Relevance*, Springer-Verlag, Berlin, Heidelberg, pp. 17–31, doi:10.1007/978-3-540-89020-1_2.

[8] Gregor Engels, Jochem M. Küster, Reiko Heckel & Luuk Groenewegen (2001): *A methodology for specifying and analyzing consistency of object-oriented behavioral models.* In: *8th European*

Soft. Eng. Conference/9th ACM SIGSOFT International Symposium on Foundations of Softw. Eng., ACM, New York, NY, USA, pp. 186–195, doi:10.1145/503209.503235.

[9] Rik Eshuis (2006): *Symbolic Model Checking of UML Activity Diagrams. ACM Trans. Softw. Eng. Methodol.* 15(1), pp. 1–38, doi:10.1145/1125808.1125809.

[10] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer & B. Nuseibeh (1994): *Inconsistency Handling in Multiperspective Specifications. IEEE Trans. Softw. Eng.* 20(8), pp. 569–578, doi:10.1109/32.310667.

[11] Bas Graaf & Arie van Deursen (2007): *Model-Driven Consistency Checking of Behavioural Specifications.* In: *Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES'07)*, IEEE, pp. 115–126, doi:10.1109/MOMPES.2007.12.

[12] Object Management Group: *UML 2.4.1 Superstructure Specification.* `http://www.omg.org/spec/UML/2.4.1`. Last accessed: 2013-12-06.

[13] Jana Koehler, Giuliano Tirenni & Santhosh Kumaran (2002): *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, Washington, DC, USA, pp. 96–.

[14] Timo Latvala, Armin Biere, Keijo Heljanko & Tommi Junttila (2004): *Simple bounded LTL model checking.* In: *In: FMCAD. Volume 3312 of LNCS*, Springer Berlin Heidelberg, pp. 186–200, doi:10.1007/978-3-540-30494-4_14.

[15] Francisco J. Lucas, Fernando Molina & Ambrosio Toval (2009): *A systematic review of UML model consistency management. Information and Software Technology* 51, pp. 1631–1645, doi:10.1016/j.infsof.2009.04.009.

[16] Zohar Manna & Amir Pnueli (1991): *Completing the temporal picture.* In: *16th Int'l Colloquium on Automata, languages, and programming*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, pp. 97–130.

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

[18] Amir Pnueli (1977): *The temporal logic of programs.* In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, IEEE Computer Society, Washington, DC, USA, pp. 46–57, doi:10.1109/SFCS.1977.32.

[19] August-Wilhelm Scheer & Markus Nüttgens (2000): *ARIS Architecture and Reference Models for Business Process Management.* In: *Business Process Management, Models, Techniques, and Empirical Studies*, Springer-Verlag, London, UK, UK, pp. 376–389.

[20] Ragnhild Van Der Straeten, Tom Mens, Ragnhild van der Straeten, Jocelyn Simmonds & Viviane Jonckers (2003): *Using description logic to maintain consistency between UML models.* In: *6th Int'l Conf. on The Unified Modeling Language, Modeling Languages and Applications*, Springer, pp. 326–340, doi:10.1007/978-3-540-45221-8_28.

[21] Aliki Tsiolakis & Hartmut Ehrig (2000): *Consistency analysis of UML class and sequence diagrams using attributed graph grammars.* In: *Workshop on Graph Transformation Systems (GRATRA)*, pp. 77–86.