Verifying Consistency between Activity Diagrams and Their Corresponding OCL Contracts

Christoph Hilken[1] Julia Seiter[1] Robert Wille[1] Ulrich Kühne[1] Rolf Drechsler[1,2]

[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{chilken,jseiter,rwille,ulrichk,drechsle}@informatik.uni-bremen.de

***Abstract*—Modeling languages such as SysML provide various description means for a precise specification of the desired system. As a system model typically uses multiple diagram types focusing on different aspects, it is crucial to keep them consistent to each other. In this paper, we propose a verification methodology which ensures the consistency between activity diagrams as blueprints for the implementation and their contracts from a block definition diagram. For this purpose, activity diagrams are transformed to OCL constraints that can be checked against pre- and post-conditions. The proposed approach is evaluated in a case study based on an industrial specification.**

I. Introduction

System descriptions based on modeling languages are becoming increasingly popular. While software developers have already been using languages such as the *Unified Modeling Language* (UML, [12]) for years, also embedded systems designers have started following this trend. In particular, the *Systems Modeling Language* (SysML, [9], [17]) is a very popular UML profile. These languages allow for a precise specification of the desired systems on an abstraction level that is appropriate for the first design steps. To this end, proper description means are provided to (formally) specify the structure and behavior of complex systems. At the same time, unnecessary and distracting implementation details which are addressed later in the design process are hidden.

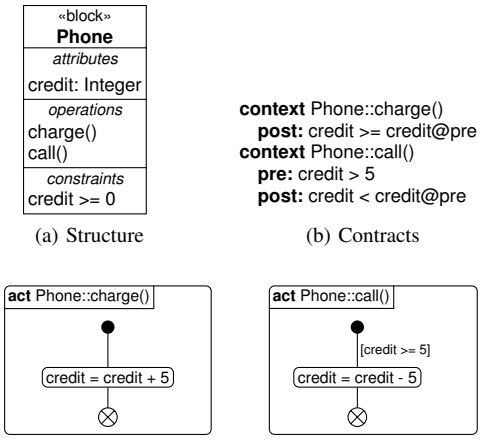
The SysML covers a variety of different diagram types. The structure of a system is usually represented by block definition diagrams specifying the available blocks, attributes, and operations as well as their relations to each other. The operations can be annotated with pre- and post-conditions which act as *contracts* for a later implementation and, by this, constitute an early formal specification of *what* the system is supposed to do. Alternatively, the functional specification of operations can also be described by state machines or activity diagrams. Activity diagrams provide a graphical representation of the control flow of an operation by making use of many imperative elements such as sequential composition, conditionals, loops, and parallel composition. They can be seen as a blueprint of *how* an operation shall be implemented.

The use of different diagram types in a single model is a common way to conveniently describe different aspects and views of a system. Moreover, such models are precise enough such that conceptional flaws introduced in this stage can already be detected – an important benefit considering the steadily increasing time-to-market constraints. However, it is a difficult problem to keep the various descriptions consistent and to avoid ambiguous or contradictory specifications.

Previous solutions addressing this problem of *inter-model consistency* mostly consider description means on the same level of abstraction (e.g. state machine vs. sequence diagrams [18]) – often by just providing syntactical rules to check the consistency [13]. Other solutions focus on static consistency (see e.g. [3], [5], [15]) or verify whether the behavior has been specified as intended (see e.g. [7], [1], [2], [14]), but only within a single description means for this purpose (e.g. a representation of behavior in terms of contracts only). In [8], an overview of further solutions is provided. However, an important question in the model-driven development of circuits and systems has not been addressed yet: Does the blueprint of an implementation provided by an activity diagram indeed satisfy the contracts defined before?

In this work, we propose a solution to this problem. We observe that, in principle, the contracts and activity diagrams already provide all details that are necessary to check their consistency. But in practice an automatic verification was not possible thus far since existing checkers rely on a formal notion of the behavior provided in the *Object Constraint Language* (OCL, [16]). Hence, a methodology is presented which transforms the activity diagrams into OCL and, by this, makes existing verification schemes applicable. A case study based on an industrial specification confirmed the usefulness of the proposed solution.

The remainder of this paper is structured as follows. Section II briefly reviews the SysML and the considered diagram types. Afterwards, Section III motivates and formalizes the problem before the proposed methodology is presented. The cornerstone of this methodology is the transformation of activity diagrams to OCL which is described in detail in Section IV. Finally, the paper concludes with a summary of the conducted case study and conclusions in Section V and Section VI, respectively.



(a) Structure (b) Contracts

(c) Activity diagrams

Fig. 1. Simple SysML model

II. Background

The *Systems Modeling Language* (SysML, [9], [17]) offers proper description means to precisely specify both the structure and the behavior of the intended system. To keep the paper self contained, this section briefly reviews the considered diagram types.

A. Specifying the Structure of the System

In SysML, the structure of a system is described by a *Block Definition Diagram* (BDD) which consists of *blocks*. Each block is composed of *compartments* that contain different information such as *attributes*, *operations*, *constraints*, or *references*. Attributes for example describe the data elements of the block, whereas operations can be used to modify them. Furthermore, constraints, so called *invariants* e.g. provided in the *Object Constraint Language* (OCL, [16]), can restrict the possible values of attributes. Different blocks can be related to each other using *associations*.

Example 1: The usage of BDDs is illustrated in Fig. 1(a). This diagram specifies a simple mobile phone consisting of a single block Phone. The phone contains an attribute credit storing the amount of money which is left in order to perform calls. The constraint in the Phone block ensures that the credit never drops below 0.

B. Specifying the Behavior of the System

The behavior of a system is given by the functionality of all available operations. Their behavior is usually specified in two steps. First, it is defined *what* an operation is supposed to do. This is done by *contracts*, e.g. by using OCL expressions that, in a pure functional fashion, describe in which system states an operation can be called and what its effects on the system state are. Afterwards, it is specified *how* an operation works. For this purpose, *activity diagrams* allow for an imperative and algorithmic specification which can be used as blueprint for an actual implementation.

1) Contracts: Contracts describe the desired behavior of an operation by a set of pre- and post-conditions. Pre-conditions specify the assumptions on the system state in which an operation shall be called whereas post-conditions describe how the state is supposed to be modified by the execution. Textual constraints, e.g. provided in OCL, are used for this purpose.

Example 2: Figure 1(b) shows the contracts that specify the behavior of all operations introduced in the BDD from Fig. 1(a). More precisely, the mobile phone application is composed of two operations:

- charge(): The charge operation increases the credit. The amount of increase is not further specified leaving room for the later implementation.
- call(): The operation call() decreases the credit by an unspecified amount for each call which is made. A call is only possible when the credit is greater than 5 which is ensured by the pre-condition.

Given these contracts, the full behavior of this (simple) mobile phone application is defined: The credit can be increased at any time and calls can only be made if the credit is greater than 5. After a call is made, the credit is decreased.

2) Activity Diagrams: An *activity diagram* can be used to graphically sketch procedures and to describe the steps which an operation is performing. By this, the behavior of an operation is described in an imperative fashion specifying how the respective operation is executed. Several graphical description means such as *actionNodes* (performing a computation), *guards* (restricting execution paths), *decisionNodes* (branching the flow), *forkNodes* (splitting the flow), etc. are available for this purpose.

Example 3: Figure 1(c) shows activity diagrams that specify an implementation of all operations introduced in the BDD from Fig. 1(a) according to the contracts from Fig. 1(b). The operation charge() is realized by a single actionNode which adds 5 to the value of credit in the Phone block. In the activity diagram specifying call(), a guard ensures that the actionNode subtracting 5 to the attribute credit is only called if credit is greater than 5. This operation is associated to the Phone block as well.

C. Specifying the Model Transformations

Model transformations are used to transform a model A of a metamodel M_A to the corresponding model B of metamodel M_B. In case of SysML, the *Meta Object Facility Query/View/Transformation* Specification (QVT, [11]) can be used to specify such transformations. QVT allows the specification of a transformation with relations (descriptive, usually bidirectional) or operational mappings (imperative, unidirectional).

Example 4: Figure 2 shows a QVT specification of a small transformation. This transformation takes a SysML model and adds the prefix string *Mobile* to the name of each block. A specification like this would transform the model from Fig. 1 into a new one in which the block *Phone* is replaced by the block *MobilePhone*.

```
transformation makeMobil(inout m:SysML)
main() {
m.objectsOfType(Block)->map makeMobil();
}

mapping inout Block::makeMobil() {
name := 'Mobile' + self.name.firstToUpper()
}
```

Fig. 2. Simple transformation

III. Verifying Activity Diagrams

Using SysML, a precise specification of a system to be implemented is provided by several different diagrams. At this early stage of the design process, it is an important task to keep these diagrams consistent with each other. Particularly BDDs and activity diagrams are of interest here, since BDDs provide information about the structure as well as operation contracts, whereas activity diagrams form a blueprint for the following implementation of the system. As activity diagrams are supposed to satisfy the contracts from the BDD, it is of great interest to ensure consistency between the two diagram types. Otherwise, the implementation will deviate from the original design intent.

Besides their use for sketching procedures with actions described in natural language, activity diagrams also allow for a *formal*, executable description. In this case, these two descriptions can be automatically checked against each other. By this, the correctness of the specified implementation can be verified right at the beginning of the design process and even before a single line of code is written – an important benefit considering the steadily increasing time-to-market constraints. In this section, first the outlined scenario is briefly illustrated by an example. Afterwards, the problem is formally defined and the proposed verification methodology is sketched.

A. Motivation

Even in the initial design steps and at the specification level, errors might occur. In fact, already the specification of the (rather simple) mobile phone application discussed in Section II contains an inconsistency.

Example 5: Consider again the specifications from Fig. 1. The activity diagrams are not consistent with all contracts. In fact, the activity diagram of the operation call() requires that credit is greater than or equal to 5 whereas the pre-condition of the same operation only requires the greater than-relation. Hence, when credit is set to 5, it should be possible to call the operation – this is not possible in the activity diagram due to the failing guard.

While this error is obviously rather simple and easy to detect, less obvious inconsistencies might be hidden in real designs. Detecting them manually is a time-consuming task. However, inconsistencies like this can also be detected automatically – even prior to the implementation process. In fact, all descriptions are provided in a formal fashion and, hence, are applicable to automatic verification methods. This is sketched in the following section.

B. Problem Formulation & General Idea

The problem illustrated above is formulated as follows: An operation o transforms a system state σ (i.e. an instance of the model) to a succeeding system state σ'. As outlined in Section II-B1, contracts restrict those states by pre- and post-conditions (denoted by $\lhd$ and $\rhd$, respectively), i.e. a transition from σ to σ' by the operation o is valid, if σ satisfies the pre-condition of o (denoted by $\lhd_o(\sigma)$) and σ' satisfies the post-condition of o (denoted by $\rhd_o(\sigma')$). At the same time, both system states σ, σ' obviously have to satisfy the invariants of the model. This is denoted by $I(\sigma)$ and $I(\sigma')$, respectively.

In a similar fashion, the execution of an operation described by an activity diagram (denoted by act) is formalized. Here, we distinguish explicitly between the initial guard condition g_o (i.e. the guard which is applied directly after the initial node) and the actual execution f_o (i.e. the symbolic representation of all actions in the activity diagram). Then, an activity diagram for an operation o transforms a system state σ to a succeeding system state σ' (denoted by $\mathsf{act}_o(\sigma, \sigma')$), if σ satisfies the initial guard of o (denoted by $g_o(\sigma)$) and σ' is consistent with the execution of o (denoted by $f_o(\sigma, \sigma')$).

On this basis, the problem considered in this work can be formalized. In order to ensure that, for an operation o, the design intent (provided by the contracts $\lhd_o, \rhd_o$) has been implemented correctly in an activity diagram act_o (with g_o, f_o), the following constraints must hold:

1) Whenever the pre-conditions $\lhd_o$ are satisfied, then the activity diagram must be executable, i.e. the guard condition g_o of the implementation must be satisfied:
$$\forall \sigma : \lhd_o(\sigma) \Rightarrow g_o(\sigma) \tag{1}$$
2) Whenever the invariants as well as pre-conditions are satisfied and the activity diagram is executed, then the succeeding system state must satisfy the invariants and the post-conditions, i.e.
$$\forall \sigma, \sigma' : I(\sigma) \wedge \lhd_o(\sigma) \wedge \mathsf{act}_o(\sigma, \sigma') \Rightarrow \rhd_o(\sigma') \wedge I(\sigma') \tag{2}$$

This problem formulation follows the established idea of *design by contract* [10].

C. Proposed Methodology

In principle, the contracts and an activity diagram already provide all details that are necessary to check for these constraints. Moreover, in the recent past effective methods for the verification of UML and/or SysML descriptions have been presented (see e.g. [7], [1], [2], [14]). However, they mainly rely on a formal notion of the behavior by means of OCL. While BDDs already provide contracts written in OCL for each operation, activity diagrams are usually not supported.

In this work, we propose to solve this problem by transforming the activity diagram into OCL and, afterwards, utilizing the resulting descriptions together with one of the approaches mentioned above to verify the two constraints. By this, the research question to be addressed in this work is reduced to:

How to transform an activity diagram into an equivalent OCL description?

IV. Transformation of Activity Diagrams

In this section, we introduce a structured approach that transforms a given activity diagram automatically into an equivalent description in terms of OCL. More precisely, since the BDD already provides contracts in the terms of pre- and post-conditions, the activity diagram is transformed to pre- and post-conditions as well. To cope with the complexity of description means which may occur in activity diagrams, the proposed methodology incorporates a divide and conquer scheme. That is, the set of description means in the considered activity diagram is subsequently reduced until a sufficiently simple representation results which can be transformed properly. We follow a two-stage approach: First, the considered activity diagram is *normalized* to the desired form; afterwards the actual *transformation* to contracts is performed. For the first step, a model transformation specified in QVT is used. However, for sake of a more inutitive description, the different mappings are described in an informal way in the following. Before we describe the above steps in detail, our general assumptions on the considered activity diagrams are discussed first. The application of the proposed approach is later illustrated by an industrial example in Section V.

A. Assumptions

In order to apply the proposed approach, a couple of assumptions have to be made on the description means which may occur in the considered activity diagrams. While these assumptions are an essential requirement for the proposed methodology, the resulting restrictions are also justified by the fact that the verification methods applied afterwards would have eventually enforced them nevertheless. The applied assumptions read as follows:

- *Formal Specification:* All descriptions must be formally specified. Although e.g. *actionNodes* in activity diagrams may also include non-formal descriptions (e.g. in natural language), those cannot automatically be transformed to unambiguous contracts.
- *Parallelism:* All parallel (sub-)operations have to be consistent with themselves. For example, *forkNode* and *joinNode* have to be used pairwise (i.e. all token generated by a *forkNode* must be joined in a *joinNode*) and parallel operations have to be independent of each other. In other words neither order nor timing aspects of parallel execution are supported. Otherwise, no unique verification result can be obtained.
- *Object Flow:* The object flow is only seen as a mapping of objects so that the control flow has to be added explicitly. This is necessary to avoid the introduction of further parallelism.
- *No Interrupts and Exceptions:* To the best of our knowledge, no formal verification approach for the modeling level is available yet that supports interrupts and/or exceptions. Verification methods that are being applied on software implementations exist, e.g. [6]. In this work only activity diagrams without interruptible regions are considered.
- *No Dynamic Loops:* The automatic verification of dynamic loops remains an open research question, and is undecidable in general. As a result, existing approaches (e.g. [4]) still rely on inputs by the designer. This contradicts the desired fully automatic approach. Hence, corresponding *loopNodes* are not supported.

B. Normalization

The first step aims at reducing the set of description means in the considered activity diagram until a representation results which is sufficiently atomic to enable a proper transformation to contracts. This normalization is conducted in five steps.

1) Flatten Hierarchy: Operations might be specified through an elaborated hierarchy of activity diagrams. Respective parts of an operation are then called e.g. through *callAction*. In order to normalize that, the whole hierarchy of an operation's description is flattened, i.e. the respective *callAction* nodes are replaced with the corresponding activity diagrams.

Example 6: Figure 3(a) shows a hierarchic activity diagram including a callAction node. This description is normalized as shown in Fig. 3(b).

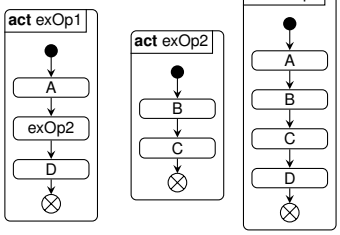
2) Resolve the Object Flow: In a next step, the object flow, i.e. *pins*, *objectNodes*, and *parameterNodes*, are removed, leaving only the control flow. For this purpose, every object flow is replaced by its source. The source of an object flow can only be an operation parameter or an attribute. Both are known inside the BDD, i.e. a simple replacement is sufficient to remove all object flows.

3) Handle Parallelism: Parallel executions are introduced through *forkNode* and *joinNode*, i.e. control nodes that split a flow into multiple parallel flows or synchronize multiple flows back to a single one, respectively. In order to derive the desired contracts, only the initial condition and the final result of an operation are relevant – not whether its actions are supposed to be executed in parallel or not.

Here, two cases have to be considered to handle this:

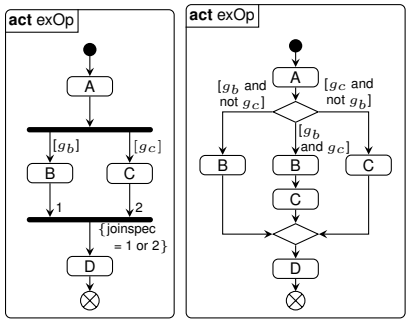
- Using the default *and*-semantic, parallel flows are replaced by one valid sequence. Since the order of actions is not relevant, any order can be chosen.
- Using another semantics, e.g. *or*-semantic in combination with guards, *forkNodes* and *joinNodes* are replaced by *decisionNodes*, respectively *mergeNodes*, with all valid guard combinations and one valid flow of actions.

Example 7: Figure 4(a) shows an activity diagram including a forkNode/joinNode-pair using or-semantics and guards. During normalization, the forkNode/joinNode is replaced by a decisionNode and a mergeNode, respectively, as shown in Fig. 4(b). Every path is representing one possible guard evaluation and the corresponding flows.



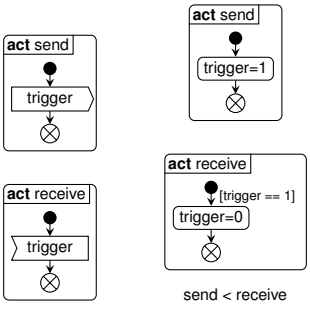
(a) Before (b) After

Fig. 3. Flatten hierarchy



(a) Before (b) After

Fig. 4. Handle parallelism



(a) Before (b) After

Fig. 5. Handle events

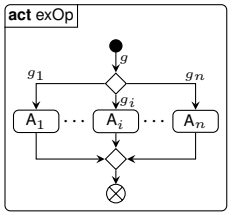
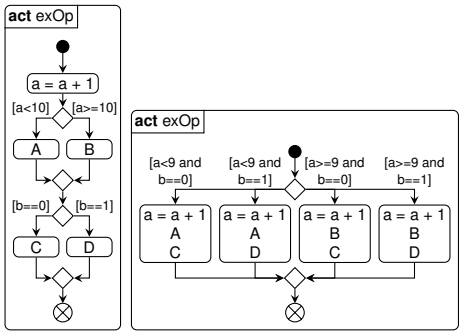
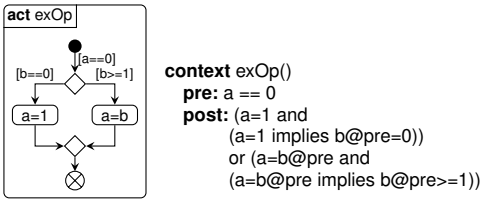


Fig. 6. Normalized



(a) Before (b) After

Fig. 7. Normalize paths



(a) Before (b) After

Fig. 8. Transformation to OCL

4) Handle Events: Events are introduced by *sendSignalAction/acceptEventAction*, i.e. control nodes that send events or wait for the occurrence of an event meeting the specified conditions, respectively. In the proposed solution, event signals are represented by attributes. Then, simple *actionNodes* and *guards* together with additional constraints on the order of execution allow for a normalization of the description.

Example 8: Figure 5(a) shows two activity diagrams, one with a *sendSignalAction* and another one with an *acceptEventAction* node. This description is normalized as shown in Fig. 5(b). Here, the corresponding event signal is represented by a global variable trigger. Then, the actionNodes and the guard equivalently represent the behavior. In addition, the order of execution is restricted by the constraints as shown in the bottom of Fig. 5(b).

5) Normalize Paths: Applying the steps from above, the resulting activity diagrams satisfies the following characteristics:

- The *initialNode* and the *flowFinalNode* simply describe the begin and the end of an operation, respectively,
- *decisionNodes*/*mergeNodes* as well as *guards* describe conditions on which actions are executed, and
- *actionNodes* describe actual computations to be performed in the operation.

As a last step, these activity diagrams are normalized, resulting in the structure shown in Fig. 6. More precisely, the nodes in the activity diagram have to be re-ordered such that all *decisionNodes* and *guards* occur in the top of the diagram, all *mergeNodes* occur in the bottom of the diagram, and all *actionNodes* occur between them. This transformation is possible by applying the following rules:

- *Merge:* If two elements of the same type occur directly after another, they are merged. Guards at *decisionNodes* and computations inside of actions are concatenated.
- *Swap and Copy:* If a *decisionNode* follows an *actionNode* or an *actionNode* follows a *mergeNode*, their positions are swapped by copying the *actionNode* into all paths.
- *Delete and Copy:* If a *decisionNode* follows a *mergeNode*, the subgraph below the *decisionNode* is copied to its corresponding *mergeNode* into every branch. Guard conditions are to be moved to the top as well.

Due to the replacement of actions when applying these rules, the guards have to be recalculated. The same is necessary when merging two actions into one.

Example 9: Figure 7(a) shows an activity diagram including several decisionNodes/mergeNodes, guards, and actionNodes. The normalized description shown in Fig. 7(b) is reached by applying the rules and recalculating the guards $a < 10$ and $a >= 10$.

C. Transformation

After the normalization, each activity diagram has the form as shown in Fig. 6. Using model to text transformations, the desired contracts are derived from this description as follows.

1) Pre-condition: The *guard* g_o at the edge between the *initialNode* and the *decisionNode* constitutes the pre-condition. Since g_o is by definition a Boolean expression, this can easily be transformed to OCL syntax.

2) Post-condition: The post-condition is constituted by the disjunction of the single branches following the *decisionNode*. More precisely, a post-condition represents the behavior of the considered activity diagram, iff for each branching *guard* g_i $(1 \leq i \leq n)$ the respective computations in the *actionNode* A_i are represented, i.e. iff $\bigvee_{g_i \in G} A_i \wedge (A_i \Rightarrow g_i)$[1] (where g_o together with all g_i and A_i form the execution f_o). For a single path in the normalized diagram, A_i evaluating to true indicates that the path on which A_i is executed has been taken. Consequently, the guard g_i of said path has to be true as well. This is expressed by the implication. Considering a disjunction over all paths in the diagram, this expresses the fact that one of the n existing paths must have been taken and that the guard of this path must be satisfied. If no A_i evaluates to true or the corresponding condition g_i evaluates to false, then none of the paths can be taken and the activity is not executable.

The respective *actionNodes* A_i represent a set S_i of statements. Since all *actionNodes* formally argue only over attributes as well as parameters and the activity diagram has been normalized as described above, each *actionNode* A_i can be transformed to $\bigwedge_{s \in S}(s.\mathrm{lhs} = s.\mathrm{rhs@pre})$, where $s.\mathrm{lhs}$ ($s.\mathrm{rhs}$) denotes the left-hand side (right-hand side) of a statement s and $@\mathrm{pre}$ refers to the frozen attribute values from the system state in which the operation is being called.

Hence, the resulting post-condition is composed as follows:

$$\bigvee_{g_i \in G} \Big(\bigwedge_{s \in S_i} (s.\mathrm{lhs} = s.\mathrm{rhs@pre}) \\ \wedge \bigwedge_{s \in S_i} (s.\mathrm{lhs} = s.\mathrm{rhs@pre}) \Rightarrow g_i@\mathrm{pre} \Big)$$

where G represents the set of all *guards* $G = \{g_1 \dots g_n\}$ at the edges following the *decisionNode*. This formula guarantees that one path in the activity diagram has been taken because it can only evaluate to true when one of the A_i evaluates to true. This again implies that the corresponding guard has to be true as well. Otherwise, the post-condition would not hold. Finally, this formula is transformed to OCL syntax.

Example 10: Figure 8(a) shows a normalized activity diagram. The pre- and post-condition derived from this description are shown in Fig. 8(b).

V. Case Study

The proposed methodology has been implemented in Java. As underlying verification technique, the solution presented in [14] has been utilized. Afterwards, the applicability of our method has been confirmed by means of an industrial specification, a SysML description of a turn indicator used in Mercedes Benz cars.

[1]This can be simplified to $\bigvee_{g_i \in G} A_i \wedge g_i$

The turn indicator offers functionality for controlling the flash signal of the car by a lever (indicating left or right) and a switch for the warning lights. Once the driver moves the lever up or down, the corresponding light is turned on and off at least three times. The light keeps flashing for as long as the lever is left in the respective position. Pushing the switch turns on both lights simultaneously. As an additional function, the warning lights can always get interrupted by the regular flash signal, i.e. when the warning lights are active and the lever is pulled, the respective flash signal is turned on instead.

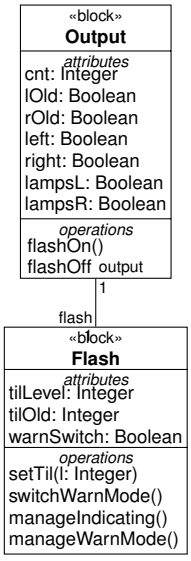
This functionality is modeled as a BDD as shown in Fig. 9(a) with the respective contracts of the operations partially shown in Fig. 9(b)[2]. The block Flash represents the controls available to the driver, i.e. the lever and the switch for the warning lights. The respective operations are employed for taking the inputs from the driver and controlling the lights of the car. The lights themselves are part of the block Output, which is responsible for switching the lights on or off depending on the driver's input. According to the contracts, flashOn() can only be called if one of the signals for turning the lights on is set and if both lights are turned off. After calling the operation, one or both lights are to be turned on. In addition to the BDD and its contracts, a blueprint for the implementation of each operation is given by activity diagrams (see Fig. 9(c) for a description of the flashOn()-operation).

Even for those rather simple descriptions, it is not obvious whether the activity diagram from Fig. 9(c) is indeed consistent to the contracts from Fig. 9(b). Manually verifying the consistency would be a time-consuming and probably error-prone task. In contrast, the proposed methodology can tackle this problem automatically in almost no time. For this purpose, the activity diagram is transformed as described in Section IV leading to the automatically generated OCL constraints shown in Fig. 9(d). Although they appear more complex and longer, they are equivalent to the activity diagram and can be used to be verified against the originally given contracts. The applied verification engine, presented in [14], is able to verify the consistency in just a fraction of a second. The same holds for all other operations in this specification.

VI. Conclusions

In this work, we have considered the verification of activity diagrams against their corresponding contracts. This represents a crucial task in model-driven development since errors in activity diagrams are likely to be propagated into the implementation. A methodology has been presented which transforms a given activity diagram to OCL and utilizes existing verification schemes for the actual comparison. A case study based on an industrial specification confirmed that, using the proposed methodology, the task can automatically be performed in almost no time.

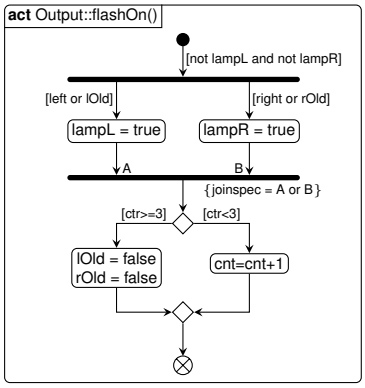
[2]For brevity, we only consider one operation, namely flashOn() in the following.



(a) BDD

context Output::flashOn()
pre: left or right or (cnt>=1 and cnt<3)
pre: not lampsL and not lampsR
post: (left@pre or lOld@pre) implies lampsL
post: (right@pre or rOld@pre) implies lampsR
post: (not left@pre and not lOld@pre and not right@pre and not rOld@pre) implies (not lampsL and not lampsR)
post: cnt@pre<3 implies cnt=cnt@pre + 1
post: cnt@pre>=3 implies cnt=cnt@pre

(b) Given Constraints



(c) Activity Diagram

context Output::flashOn()
pre: not lampsL and not lampsR
post:
((lampsL and cnt=cnt@pre + 1)
and ((lampsL and cnt=cnt@pre + 1)
implies ((left@pre or lOld@pre) and
not (right@pre or rOld@pre) and cnt@pre<3)))
or
((lampsL and not lOld and not rOld)
and ((lampsL and not lOld and not rOld)
implies ((left@pre or lOld@pre) and
not (right@pre or rOld@pre) and cnt@pre>=3)))
or
((lampsR and cnt=cnt@pre + 1)
and ((lampsR and cnt=cnt@pre + 1)
implies (not (left@pre or lOld@pre)
and (right@pre or rOld@pre) and cnt@pre<3)))
or
((lampsR and not lOld and not rOld)
and ((lampsR and not lOld and not rOld)
implies (not (left@pre or lOld@pre)
and (right@pre or rOld@pre) and cnt@pre>=3)))
or
((lampsL and lampsR and cnt=cnt@pre + 1)
and ((lampsL and lampsR and cnt=cnt@pre + 1)
implies ((left@pre or lOld@pre)
and (right@pre or rOld@pre) and cnt@pre<3)))
or
((lampsL and lampsR and not lOld and not rOld)
and ((lampsL and lampsR and not lOld and not rOld)
implies ((left@pre or lOld@pre)
and (right@pre or rOld@pre) and cnt@pre>=3)))

(d) Generated OCL

Fig. 9. Specification of a turn indicator

ACKNOWLEDGMENTS

We would like to thank Mathias Soeken for fruitful discussions. This work was supported by the Graduate School SyDe (funded by the German Excellence Initiative within the University of Bremen's institutional strategy), the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001, as well as the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1 and a research project under grant no. WI 3401/5-1.

REFERENCES

[1] A. Baruzzo and M. Comini. Static Verification of UML Model Consistency. In *Proc. 3rd Workshop Model Design and Validation*, pages 111–126, 2006.

[2] J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL Operation Contracts. In M. Leuschel and H. Wehrheim, editors, *Integrated Formal Methods*, volume 5423 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2009.

[3] M. Gogolla, M. Kuhlmann, and L. Hamann. Consistency, Independence and Consequences in UML and OCL Models. In *Test and Proof*, pages 90–104. Springer, 2009.

[4] A. Ireland and J. Stark. On the Automatic Discovery of Loop Invariants. In *NASA Conference Publication*, pages 137–152, 1997.

[5] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler. Formalizing UML Models and OCL Constraints in PVS. *Electronic Notes in Theoretical Computer Science*, 115:39–47, 2005.

[6] X. Li, J. Hoover, and P. Rudnicki. Towards Automatic Exception Safety Verification. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 396–411. Springer, 2006.

[7] X. Li, Z. Liu, and J. He. Consistency checking of UML requirements. In *International Conference on Engineering of Complex Computer Systems*, pages 411–420. IEEE Computer Society, 2005.

[8] F. J. Lucas, F. Molina, and A. Toval. A Systematic Review of UML Model Consistency Management. *Information and Software Technology*, 51(12):1631–1645, 2009.

[9] G. Martin and W. Müller. *UML for SOC Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[10] B. Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992.

[11] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. *Version 1.1*, 2011.

[12] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. Addison-Wesley Longman, Essex, UK, 1999.

[13] P. G. Sapna and H. Mohanty. Ensuring Consistency in Relational Repository of UML Models. In *International Conference on Information Technology*, pages 217–222, 2007.

[14] M. Soeken, R. Wille, and R. Drechsler. Verifying Dynamic Aspects of UML Models. In *Design, Automation and Test in Europe*, pages 1077–1082, 2011.

[15] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using Boolean satisfiability. In *Design, Automation and Test in Europe*, pages 1341–1344, 2010.

[16] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise modeling with UML*. Addison-Wesley Longman, Boston, MA, USA, 1999.

[17] T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[18] X. Zhao, Q. Long, and Z. Qiu. Model Checking Dynamic UML Consistency. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 440–459. Springer, 2006.