Formulating Model Verification Tasks Prover-Independently as UML Diagrams

Martin Gogolla[1], Frank Hilken[1], Philipp Niemann[2], and Robert Wille[2,3]

[1] University of Bremen, Bremen, Germany
{gogolla|fhilken}@informatik.uni-bremen.de, philipp.niemann@dfki.de
[2] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
[3] Johannes Kepler University, Linz, Austria
robert.wille@jku.at

Abstract. The success of Model-Driven Engineering (MDE) relies on the quality of the employed models. Thus, quality assurance through validation and verification has a tradition within MDE. But model verification is typically done in the context of specialized approaches and provers. Therefore, verification tasks are expressed from the viewpoint of the chosen prover and approach requiring particular expertise and background knowledge. This contribution suggests to take a new view on verification tasks that is independent from the employed approach and prover. We propose to formulate verifications tasks in terms of the used modeling language itself, e.g. with UML and OCL. As prototypical example tasks we show how (a) questions concerning model consistency can be expressed with UML object diagrams and (b) issues regarding state reachability can be defined with UML sequence diagrams.

1 Introduction

Software development following the *Model-Driven Engineering* (MDE) paradigm focuses on models in contrast to traditional code-centric approaches. Models are said to offer advantages like a high degree of abstraction or platform-independence. As models become the central artifacts – particular in early stages of the design process – means for model quality assurance in form of validation ("Are we building the right product?") and verification ("Are we building the product right?") become indispensable. In particular, verification techniques get more and more important, since they allow to check whether a system to be realized is described and behaves as intended before a single line of programming code is written.

Nowadays, the UML *(Unified Modeling Language)* and the OCL *(Object Constraint Language)* are frequently applied modeling languages. Correspondingly, a substantial number of verification techniques has been developed for models provided in UML/OCL. The spectrum of approaches ranges from solutions for structural and behavioral verification tasks as well as along the employed verification engines such as theorem provers [8], solvers for *Constraint Satisfaction Problems* (CSP) [9]), Petri nets [10], model checkers [18], intermediate languages

like *Alloy* or *Kodkod* [1, 28], or solvers for *Boolean satisfiability* (SAT) and *SAT Modulo Theories* (SMT) [26, 27].

However, these approaches often address the respective verification tasks from their own particular perspective and with respect to their paradigms. For example, solutions based on SAT or SMT require a description of the considered verification tasks in terms of propositional logic or bit-vector logic, respectively. This poses a significant challenge to the designer, since expertise and background knowledge about the employed verification approach and the used tool is needed in order to formulate a verification task. Moreover, this nullifies several of the benefits of using UML/OCL models such as the easy accessibility of a system description also for non-technical stakeholders, the high-degree of freedom, as well as the independence from programming or, in this case, verification languages.

In this work, we propose a solution to this problem by introducing a view on verification tasks in a tool- and approach-independent manner. The main idea is as follows: Instead of formulating the respective verification task in a tool-related language (such as propositional logic or bit-vector logic), we propose to describe them in terms of the used modeling language (such as UML/OCL) itself. To this end, we consider some well-known and frequently applied verification tasks such as consistency or reachability and provide corresponding formulations in UML/OCL.

Overall, this allows designers to formulate additional properties for an existing UML/OCL model which are not explicitly part of the system description, but represent verification tasks. By this, designers can formulate verification tasks with description means they are most familiar with and in which they designed the currently considered model anyway.

The structure of the rest of this paper is as follows. Section 2 introduces central notions and the paper's background. Section 3 shows how a representative verification tool for UML/OCL models currently handles verification tasks. In Sect. 4, the central idea of this work, namely the formulation of structural verification tasks within UML/OCL is introduced and illustrated. Section 5 concentrates on behavioral verification tasks from the developer's point of view and discusses advantages of our proposal. Finally, related work is discussed in Sect. 6 before the paper is concluded in Sect. 7.

2 Preliminaries and Background

Modeling languages such as the UML have been established to specify the design of complex systems. They provide a broad variety of different concepts such as class diagrams, sequence diagrams, or activity diagrams which are expressive enough to formally specify a complex system – especially together with textual constraints, e.g., in terms of OCL. These formal descriptions additionally allow for the verification of the respective specification already in the absence of a specific implementation, i.e., in an early stage of the design where flaws can be eliminated at relatively low costs.

The corresponding verification tasks can be divided into *Structural Verification Tasks*, where single states of the system are considered, as well as *Behavioral Verification Tasks*, where sequences of system states together with the connecting transitions (e.g., described in terms of state charts or operations with pre- and postconditions) are considered, see e.g., [16] for an overview.

A very common structural verification task is to check the *consistency* of a model, i.e., investigating whether the model description is consistent in the sense that an instantiation of the model exists which satisfies all of the model constraints. For behavioral verification, a typical task is to consider the *reachability* of certain good or bad states from a given initial state. As these two are very popular verification tasks, they will be considered as stereotypes in the remainder of the paper to illustrate existing approaches and the proposed concepts. Other verification tasks include, for instance, to check whether (1) the model satisfies certain properties such that corresponding constraints hold for any instantiation of the model, or to check whether (2) the invariants are independent or possibly imply each other (e.g., in order to find a minimal set of invariants or constraints) [13]. Typically either solely structural or solely behavioral aspects are considered, though there are a few works that consider the model structure and behavior at once (e.g., by considering operation contracts in combination with invariants) [15].

For both categories of verification tasks, a variety of automatic solving approaches have been introduced. The main idea of most of these approaches is to encode verification problems in a language that can be passed to a dedicated solving engine and transfer the results (more or less) back to the level of UML and OCL. In this context, different languages and solving engines have been proposed such as approaches (a) using *theorem provers* like Isabelle [8], (b) reformulating the problem as a *Constraint Satisfaction Problem* (CSP) [9], (c) using *Petri nets* [10], (d) addressing *model checkers* [18], (e) using intermediate languages like *Alloy* or *Kodkod* [1,28] though finally resulting in a *SAT* problem, or (f) using a direct encoding in the more general language of *SMT* [26,27].

All these approaches have their very own characteristics and need a high amount of expert knowledge and specific experience in order to be used which cannot be expected from a common developer. And, many approaches often support a single or a small set of verification tasks only, such that a developer would need to familiarize with many of these approaches in order to conduct a reasonable variety of verification tasks. This problem will be illustrated in more detail in Sect. 3 where we will show how two popular verification tasks (consistency and reachability) are formulated in one of these approaches.

In order to show examples for verification tasks and also to illustrate the proposed new concepts, we will make use of a running example as depicted in Fig. 1. The model describes a `CivilStatusWorld` with persons having a gender and a civil status attribute and marriages between persons determined by a reflexive association. Operations for marrying and divorcing are provided as well as a query operation `spouse` determining a (possible) set of persons. Under the assumption that the model is bigamy-free, this operation returns a singleton set.

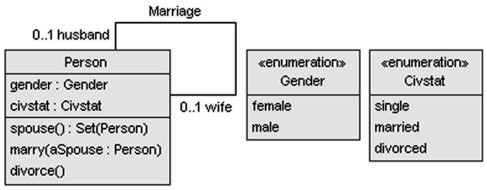

Fig. 1. Example class diagram.

OCL invariants and operation contracts in form of OCL pre- and postconditions as shown in Fig. 2 further restrict the structural and behavioral aspects of the model. More precisely, the query operation `spouse` is defined in a defensive way as a set-valued operation because the class diagram allows a person to have both a wife and a husband. The aim of the model is to have the empty set or a singleton set as the result for `spouse`, if all multiplicities and the invariant hold. The invariant establishes a connection between the gender attribute and the marriage role names as well as a connection between the `civstat` attribute and the `spouse` operation. We only show the contract for the operation `marry`, because the `divorce` contract is formulated analogously. `marry` has one precondition and a first ordinary postcondition. The second `marry` postcondition is a frame condition that explicitly requires that (a) `marry` does not introduce new objects nor change the gender attribute and (b) except the `spouse` set and except the self object (on which `marry` is called) all other objects are left unchanged w.r.t. the roles and the remaining attributes.

When one wants to enable verification of contract behavior, one must either have an operation implementation (in that case the verification results are relative to the given implementation) or one must say in a declarative, complete way what the effect of an operation is, in particular, which things (attributes or roles from the class diagram) are changed by the operation and which things are left unchanged. In the example, the two postconditions serve this purpose.

3 Manifesting Consistency and Reachability in Tools

We now explain three typical verification tasks for the running example and show how they are realized in one UML and OCL tool. The discussion underpins our claim that specific knowledge and expertise is needed for successfully verifying properties in UML and OCL models. The three verification tasks are as follows.

1. Assume a system state is given with (a) objects possessing partially specified attribute values, (b) one female person participating in a marriage (where the roles wife and husband are left unspecified) and (c) another present male person. The first verification task now asks whether the system state can be completed to a full object diagram. If successful, this would prove consistency of the structural model, i.e., satisfiability of the class diagram including the multiplicities and the invariants.

```
spouse():Set(Person)=
  if wife->notEmpty and husband->notEmpty then
    Set{wife,husband} else
  if wife->notEmpty then
    Set{wife} else
  if husband->notEmpty then
    Set{husband}
  else
    Set{}
  endif endif endif

context Person inv traditionalRoles:
  ( gender=#female implies wife->isEmpty ) and
  ( gender=#male implies husband->isEmpty ) and
  ( spouse()->notEmpty = (civstat=#married) )

context Person::marry(aSpouse:Person)
pre unmarriedDifferentGenders:
  self.spouse()->isEmpty and aSpouse.spouse()->isEmpty and
  Set{self.gender,aSpouse.gender}=Set{#female,#male}
post married:
  Set{aSpouse}=self.spouse() and Set{self}=aSpouse.spouse() and
  self.civstat=#married and aSpouse.civstat=#married

post personUnchangedExceptSet:
  let x=self.spouse()->including(self) in
  Person.allInstances@pre=Person.allInstances and
  Person.allInstances->forAll(p|
    (p.gender@pre=p.gender) and
    (x->excludes(p) implies p.civstat@pre=p.civstat) and
    (x->excludes(p) implies p.wife@pre=p.wife) and
    (x->excludes(p) implies p.husband@pre=p.husband))
```

Fig. 2. OCL query operation, invariant, and operation contract.

2. The second verification task asks whether it is possible to construct a system state with a marriage link where both participating persons have the same gender that is however not a priori fixed. If this is not possible, then the fact that in a marriage the participating persons must have different genders is a consequence of the model.
3. The third verification task checks whether it is possible to find a sequence of operation calls that leads from four single persons to four married persons. In case all operations, i.e., both marry and divorce, show up, this would show the satisfiability of the invariants considered together with the operation contracts. It would guarantee the satisfiability of the structural model (class diagram with multiplicities and invariants) considered together with the behavioral model (operation contracts).

We classify tasks (1) and (2) as consistency problems (because they aim at constructing one consistent system state) and task (3) as a property reachability problem (because a particular property must be reached when starting in a given initial situation). A general classification of verification tasks was proposed recently in [16].

USE (Uml-based Specification Environment) is a modeling tool for a subset of UML and for full OCL [11,12]. USE offers options to validate and verify UML and OCL models, in particular by employing a component called model validator

that is able to automatically construct object diagrams for UML class diagrams enriched by OCL invariants [12].

The first verification task is realized in USE by expressing the verbally expressed requirements as an additional OCL constraint and by employing the USE model validator. In general, the model validator considers a UML and OCL model with invariants together with a so-called configuration providing bounds for the possible object diagrams that are to be constructed (configuration examples can be found in [12]). The configuration bounds determine finite populations of classes, associations, datatypes and attribute values. The model validator then tries to construct an object diagram satisfying the class diagram and the invariants under the stated bounds. In this case, the additional constraint requires three persons to exist with particular attribute values and particular association participation conditions as stated below.

```
context Person inv VerificationTask1:
Person.allInstances->exists(A,B,C |
  Set{A,B,C}->size=3 and
  A.gender=#female and
  ( (A.husband=B and B.wife=A) or (B.husband=A and A.wife=B) ) and
  C.gender=#male)
```

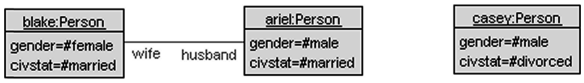

Fig. 3. USE Solution for first verification task.

As shown in Fig. 3, the model validator is successful in finding a fitting object diagram. Thus the first verification task is mastered, and the consistency of the model has been proven.

For the second validation task another OCL invariant is loaded in addition to the present model invariant. The invariant is stated below. In particular, this invariant requires a marriage between two persons which possess the same gender attribute value.

```
context Person inv VerificationTask2:
Person.allInstances->exists(P1,P2 |
  Set{P1,P2}->size=2 and
  (P1.wife=P2 or P1.husband=P2) and
  P1.gender=P2.gender )
```

In this case, the model validator reports that the model is unsatisfiable, i.e., no valid object diagram can be found. From this fact we conclude that the additional requirement involving two persons with the same gender in a marriage cannot be satisfied, and that thus the gender attributes values in a marriage must be different.

For the third verification task, the stated USE model is first transformed into a so-called filmstrip model [14]. In the filmstrip model, additional classes and associations are introduced that serve for representing a sequence of object

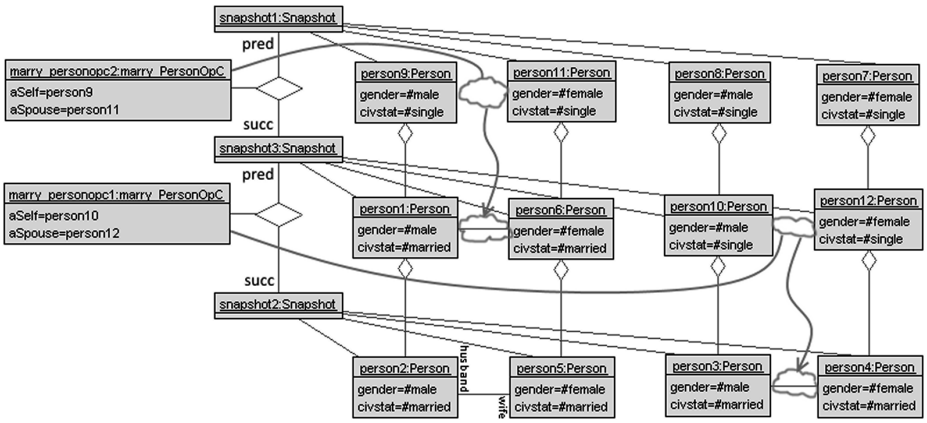

Fig. 4. USE Solution for third verification task (4 persons single to married).

diagrams from the originally stated model within a single object diagram; the original sequence of object diagrams becomes a sequence of (so-called) snapshot objects with operation call objects connecting them, as shown in Fig. 4. Additional OCL constraints guarantee that the filmstrip model behaves properly, for example, that the snapshot objects are not linked in a cyclic way.

The requirements from the third verification task that initially all persons are single and all persons are finally married are expressed as OCL invariants on the filmstrip model as stated below. The subexpressions involving `any` select the first, resp. last snapshot. The configuration for this verification task allows up to four operation calls, either marry or divorce calls, and the four person demand is reflected by appropriate settings for the number of objects in class `Person` (and could be restricted even more by another invariant).

```
context Person inv allInitiallySingle:
  Snapshot.allInstances->any(s | s.pred()=null).person->
    forAll(p | p.civstat=#single)

context Person inv allFinallyMarried:
  Snapshot.allInstances->any(s | s.succ()=null).person->
    forAll(p | p.civstat=#married)
```

The solution for this verification task is shown in Fig. 4. This filmstrip object diagram with two operation call objects and three snapshot objects corresponds to a sequence diagram in the original (application) model with two operation calls dealing implicitly with three object diagrams: one object diagram before the first call, one between the two calls, and one after the second call. These three (implicit) object diagrams are made explicit in Fig. 4 through the three snapshot objects.

4 Viewing Verification Tasks as UML Diagrams

This section is devoted to the explanation how verification tasks can be developed and represented in a way that is independent from an underlying proving engine. As good UML "citizens", we believe that UML can be employed for a lot of tasks within the software development process. The main idea of UML is to represent issues and artifacts independent of a needed underlying (proving) engine. The idea that we want to contribute is to allow non-verification experts to phrase requirements and formal properties with UML, the language they use anyway to express their models. Even verification experts may find this attractive.

We will go through the verification tasks and present UML diagrams for them. The first task about UML and OCL model consistency can be graphically shown as the object diagram in Fig. 5. The elements from the verbal explanation are translated into respective UML features, basically objects, links possessing role names and association names as well as attribute values. In some spots, concrete values (as e.g., `#female`) or concrete items from the class diagram (as e.g., Person or Marriage) are shown. Now, the idea, that we come up with, is not only to allow concrete items in a UML diagram, but to indicate some "open", not yet fixed items that represent placeholders that are to be filled by an underlying engine. Such a UML diagram with placeholders may be seen as a UML query (stealing ideas from QBE [30]) or a verification task expressed in UML. In order to distinguish between concrete items and placeholders, placeholders are syntactically marked with a starting question mark. In principle, every spot in a UML diagram, where some concrete item may be written down, may also be filled with a placeholder. In the example we have used placeholders for attribute values and role names.

The task for the underlying proving engine is now to show substitutions or answers for the placeholders with suitable concrete items. The developer will typically have a particular expectation for the possible answers. In the example, this could be that `?RA` can only be substituted with the role wife, whereas `?CC` could be replaced by one of `#single` or `#divorced` or even `#married` if it is allowed to add Person objects and Marriage links.

The second verification task can be graphically presented with the object diagram in Fig. 6. Here, the placeholder `?G` is used in two different spots, for the gender attribute value of persons `P1` and `P2` which are required to be connected by a marriage link. This expresses that the two persons in the marriage possess the same gender.

The third verification task is represented as a sequence diagram in Fig. 7. Four lifelines represent objects, two OCL constraints express an initial and a final condition and placeholders are used for operation calls. The standard UML sequence diagram features loop and alt (for alternative) are used to formulate that an operation call can go to one of the four objects and that a sequence of such calls is allowed.

In Fig. 8 we show a sequence diagram with a solution for the third task. This solution corresponds to the USE filmstrip object diagram from Fig. 4. Thus in general, it is desirable to see a found solution not only on the level of

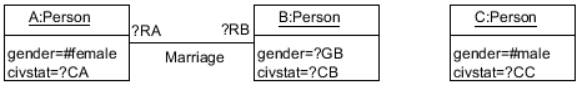

Fig. 5. First verification task as UML diagram.

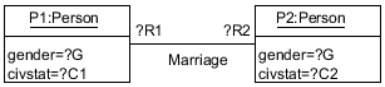

Fig. 6. Second verification task as UML diagram.

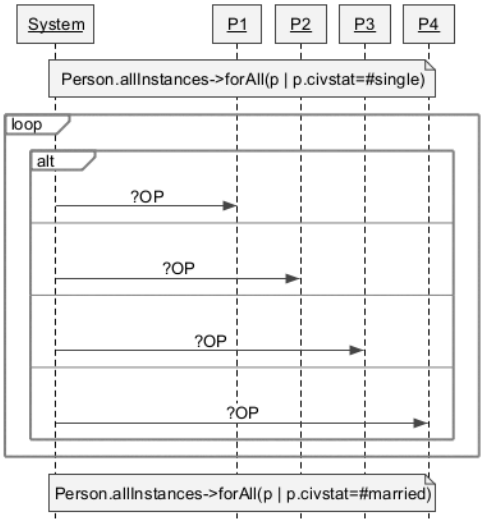

Fig. 7. Third verification task as UML diagram.

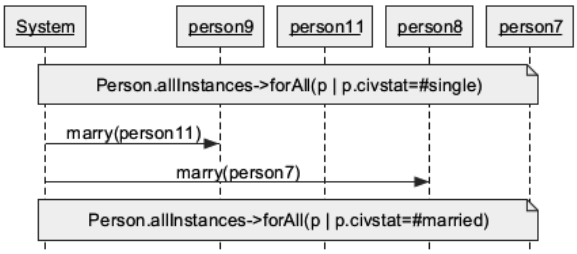

Fig. 8. Solution for third task viewed as UML sequence diagram.

the employed technology, but it is necessary to transform a found solution onto the level on which the verification task was formulated. In the concrete case, the transformation process from the filmstrip object diagram to the sequence diagram can be automated. The solution for task 1 was already shown as a UML object diagram in Fig. 3, whereas for task 2 no UML diagram was shown, because that task was not satisfiable.

Let us shortly wrap up and look back at the verification task formulation in UML and the proposed use of placeholders. The above examples employ the following UML diagram and OCL features for task formulation: in the object diagram we had objects, associations, roles, attribute values, and OCL formulas; in the sequence diagram we saw lifelines for objects, operation calls, alternative calls, calls within a loop, and on lifelines closed OCL formulas (or on a lifeline there could be a partial or complete object diagram as well). These language features have a precise meaning for task formulation. The proposed verification tasks can be transformed into prover-specific approaches. For our three example verification tasks, a precise meaning is given by the added OCL constraints before starting the verification process. These constraints can be retrieved from the graphical verification task representation automatically. As said already, in principle, we do not see any reason to restrict UML language features for task formulation as long as a precise task is determined.

Essential for our proposal is the use and role of placeholders and the kind of UML diagram features that placeholders can stand for. Currently we have had in the examples placeholders for the following features: attribute values, roles in associations, association names (not in the running example, as there is only one association in the example class diagram), operation calls and implicit or explicit operation parameters. The intention of placeholders is that they will be replaced by the underlying analysis or proving engine with concrete UML "items". These UML items either may come from the model (e.g., the class diagram) or may be explicitly provided by the developer as currently stated in a USE configuration.

5 A Developer's View on Verification Tasks

So far, we have demonstrated the basic idea of prover-independent verification tasks and sketched how they can be represented with diagrams. We now want to focus on the simplifications for developers in regards to the knowledge required to express verification tasks that can be used in any verification tool. With the tasks being tool-independent, the developer is not required to have any specific further knowledge about the tools. Even complex behavioral verification tasks can be formulated with mostly intuitive UML and OCL language features alone.

In order to illustrate the simplifications for developers, we consider the model of a traffic light which is depicted in Fig. 9. As we will see in the following, the model exhibits significantly more interesting behavioral aspects than the `CivilStatusWorld` model considered above.

The main component of the traffic light is the `Controller` which is connected to exactly one (visual) signal for cars and exactly one visual and acoustic signal

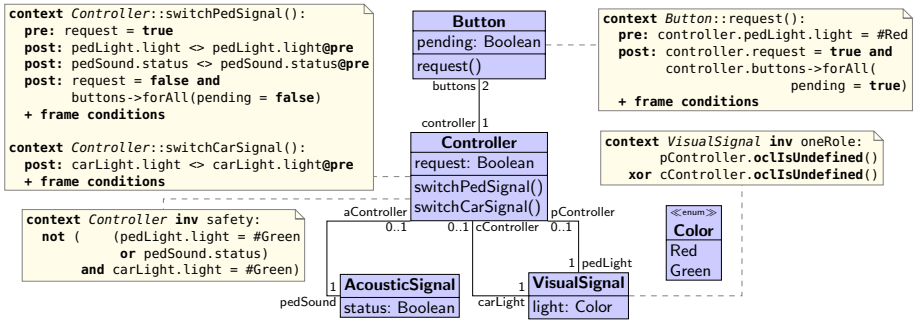

Fig. 9. Traffic light example

each for the pedestrians. Two buttons are connected to the controller that can be pushed in order to indicate a pedestrian crossing request to the controller. When one button is pushed, all connected buttons indicate that there is a pending request. The invariant `safety` states a general safety property for traffic lights, i. e., that both the signals for the pedestrians and the signal for the cars must not indicate a safe crossing at the same time. Finally, the invariant `oneRole` ensures that each visual signal can either serve as a signal for pedestrians or cars, but not both at the same time.

This model allows for instantiating various meaningful states that are vital for operating the system, but also several states that shall never be reached in practice. For instance, the standard idling state of the system in which a green light is shown to the cars while there is no pending request for a pedestrian crossing is shown on the top of Fig. 10. Below it, we can see a (partially defined) state where all visual signals are turned red while the acoustic signal indicates a safe crossing for pedestrians. Though this state is not violating any of the model's invariants, it shall never be entered in practice. Thus, it should just be unreachable by the definition of the operations. As it does not make a difference for the rejection of the state whether there is a pending request or not, unnamed placeholders are used to express that the values of the corresponding attributes are insignificant.

Now, the designer might be interested to find out whether certain states are reachable from the standard idling state. Using the proposed diagram-based approach, this task can be formulated very comfortably by (1) specifying the (partial) system states that shall serve as the start/target of the reachability analysis in terms of object diagrams and (2) employing them in a sequence diagram that allows arbitrary behavior in order to reach the target, i.e., a loop of arbitrary alternative operation calls on arbitrary objects. The corresponding diagram is shown in Fig. 11. Note that lifelines and objects for signals are not shown as no operations can be called on them.

While this formulation will fortunately yield UNSAT (proving that the erroneous state is not reachable within the number of steps that are specified as the upper limit of iterations of the loop), the designer might want to find out

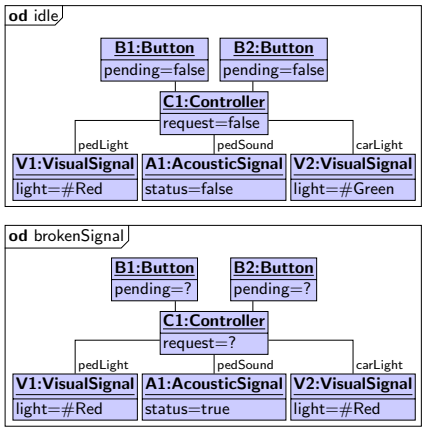

Fig. 10. (Partial) system states for the traffic light model.

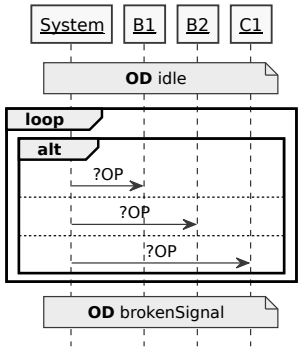

Fig. 11. Simple reachability formulation

whether it is possible to reach a state that enables a safe crossing for pedestrians and, at the same time, allows for returning to the idling state afterwards. This task can be formulated by extending the existing diagram with another loop of arbitrary operation calls (for returning to the initial state) and an intermediate state (which expresses the safe crossing for pedestrians) as shown in Fig. 12. The corresponding partial object diagram for the intermediate state is shown in Fig. 13. Note that the state only requires the visual signal for the pedestrians to show a green light, while the assignment of all other attributes are left open.

While this formulation will unfortunately also yield UNSAT (proving that there is no possibility to reach a green light for pedestrians and return to the idle state afterwards), the designer may query whether the intermediate state is reachable at all. This can be formulated by simply replacing the target state – more precisely, its object diagram – in the first sequence diagram with this state. Then, the formulation will yield that the intermediate state is indeed reachable from the idle state (e.g., using the operation call sequence `B1.request()`, `C1.switchCarSignal()`, `C1.switchPedSignal()`), but interchanging start and target state will show that this is not true for the way back, i.e., the idle state is not reachable from the intermediate state. One reason for this could be, that the intermediate state – more precisely, all states for which the visual signal for the pedestrians shows a green light – are deadlocks. To verify this property, a formulation as in Fig. 14 can be employed which asks whether any operation can be called in a (partially) given system state at all. In this special case, the operations `B1.request()` and `C1.switchPedSignal()` may not be called, since their preconditions are not fulfilled. In addition, calling `C1.switchCarSignal()` would yield a situation where both the lights for pedestrians and cars are green, which violates the safety invariant. Consequently, the state characterization in Fig. 13 indeed describes a deadlock scenario. Note that, in contrast to the pre-

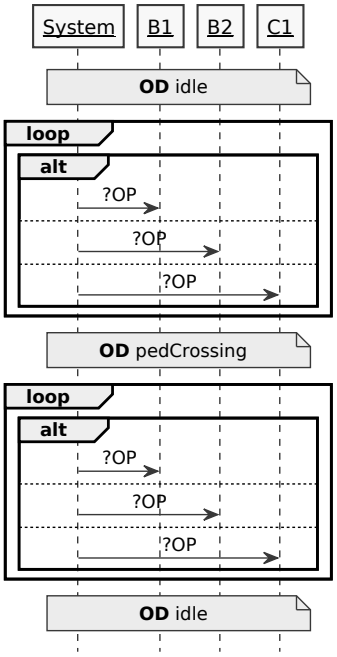

Fig. 12. Extended reachability formulation

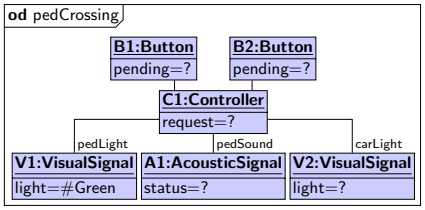

Fig. 13. Deadlock states

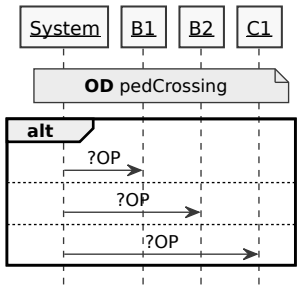

Fig. 14. Deadlock check

vious diagrams, there is no more loop in Fig. 14 and no restrictions are applied to the succeeding system state.

To summarize, this small case study shows that a wide spectrum of behavioral verification tasks can be formulated independently of a specific prover technology in terms of a modeling language, here UML-like sequence diagrams with some additional, new features.

6 Related Work

There is related work sharing the aim of this paper to improve the usability of UML/OCL verification techniques. Concerning the question of building a user-friendly interface, much work has, e.g., been done in the theorem prover community (see e.g., [2, 17, 19, 20]). There seems to be a high awareness in that community for the need to improve the usability of provers [6]. On the one hand, this is an implicit requirement as using theorem provers requires much interaction by the user. On the other hand, it is recognized that these provers focus on assisting particularly trained and skilled users, while they are difficult to use for non-expert users. Consequently, user-friendly interfaces play an important role in promoting the benefits of the underlying techniques [2].

However, these approaches do not aim at hiding details of the underlying verification technique, but only provide a “nicer”, graphical interface. To this end,

the potential of using UML diagrams to hide details and, thus, make verification easier accessible, has been recognized. UML diagrams have been used for various purposes so far: many verification approaches present solutions (witnesses found by the prover) back at the level of the model, e.g., in terms of UML diagrams. In contrast, in [23] UML sequence diagrams are used for visualizing patterns of temporal logic as part of a UML-based front-end to a formal verification/model checking toolset. More precisely, the sequence diagrams depict the desired behavior which the user can select from and combine in order to tailor dedicated verification tasks (formulas). In [21], it is suggested to combine UML diagrams and the B formalism in a design flow for hardware. However, the user can hardly specify particular properties to be verified.

The general and nice idea to employ placeholders in query languages is due to QBE [30]. A combination of model checking but having in mind a particular application for business processes is proposed in [3]. Placeholders, partly also with similar notation as here, have been used in the study of class model patterns and anti-patterns [5, 7], in the consideration of general model quality [4] and in the context of domain-specific languages [22]. Recently, there has been a proposal for a "user-friendly" interface to Alloy which employs a similar idea to formulate verification tasks by modelling them graphically [29] focusing on structural verification. In [24, 25] so-called partial models are put forward, a general framework being less tuned to specific verification tasks as we want to cover.

So far UML diagrams in combination with OCL expressions have not been used as a means for formulating dedicated structural and behavioral verification tasks.

7 Conclusion

This contribution proposed to formulate model verification tasks from the viewpoint of the employed modeling language. We aim to relieve the developer from expressing tasks only on the basis of the used approach and proving engine. Our proposal aims at giving non-verification experts the option to work with formal verification approaches. We have used UML object diagrams to formulate consistency issues and UML sequence diagrams for reachability topics. Central in our approach are so-called placeholders representing open items that can occur in UML diagrams and that should be substituted by model elements. Solutions in terms of substitutions and thus verification task feedback should be given in terms of the employed modeling languages as far as possible, e.g., as UML diagrams.

We have concentrated here on prover-independent task formulation. As one topic for future work we identify the open details for the transformation into prover- and approach-specific task formulation. The expressibility of the approach, i.e,. the answer to the question which verification tasks can be formulated by UML diagrams, is bounded on the one hand by the employed verification task features, but on the other hand it is an open question how to enable the formulation of all possible verification tasks in general by the modeling language itself.

Our proposal has to be consolidated and validated by an implementation of what has been sketched by the features that we used for task formulation. Other UML diagram kinds than object and sequence diagrams, in particular communication diagrams with communication channels and state machines for attribute, role or OCL expression evolution have to be studied in more detail. Good explanations in the case of unsatisfiable tasks indicating the "guilty" model parts or at least identifying the "innocent" model parts have to be developed. Guilty model parts, i.e., the parts that essentially contribute to the invalidity could be any model element, e.g., classes, associations, invariants, contracts or even more detailed information, for example, subformulas of constraints. Last but not least, larger case studies should give feedback on the practicability of the proposal.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: Model Driven Engineering Languages and Systems, pp. 436–450. Springer (2007)
2. Arshad, F., Mehmood, H., Raza, F., Hasan, O.: g-hol: A graphical user interface for the HOL proof assistant. In: Formal Techniques for Safety-Critical Systems, FTSCS 2015. pp. 265–269 (2015)
3. Awad, A., Sakr, S.: On efficient processing of BPMN-Q queries. Computers in Industry 63(9), 867–881 (2012)
4. Balaban, M., Maraee, A., Sturm, A., Jelnov, P.: A pattern-based approach for improving model quality. Software and System Modeling 14(4), 1527–1555 (2015)
5. Ballis, D., Baruzzo, A., Comini, M.: A minimalist visual notation for design patterns and antipatterns. In: 5th Int. Conf. Information Technology: New Generations (ITNG 2008). pp. 51–56 (2008)
6. Beckert, B., Grebing, S.: Evaluating the usability of interactive verification systems. In: Proc. 1st Int. Workshop Comparative Empirical Evaluation of Reasoning Systems. pp. 3–17 (2012)
7. Bottoni, P., Guerra, E., de Lara, J.: A language-independent and formal approach to pattern-based modelling with support for composition and analysis. Information & Software Technology 52(8), 821–844 (2010)
8. Brucker, A.D., Wolff, B.: HOL-OCL: A formal proof environment for UML/OCL. In: Fiadeiro, J.L., Inverardi, P. (eds.) Fundamental Approaches to Software Engineering, 11th Int. Conf., FASE'2008. LNCS, vol. 4961, pp. 97–100. Springer (2008), http://dx.doi.org/10.1007/978-3-540-78743-3_8
9. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: First International Conference on Software Testing Verification and Validation, ICST 2008. pp. 73–80. IEEE Computer Society (2008)
10. Choppy, C., Klai, K., Zidani, H.: Formal Verification of UML State Diagrams: A Petri Net based Approach. Softw. Eng. Notes 36(1), 1–8 (2011)
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
12. Gogolla, M., Hilken, F.: Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In: Oberweis, A., Reussner, R. (eds.) Proc. Modellierung (MODELLIERUNG'2016). pp. 203–218. GI, LNI 254 (2016)

13. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) Proc. 3rd Int. Conf. Test and Proof (TAP'2009). pp. 90–104. Springer, Berlin, LNCS 5668 (2009)
14. Hilken, F., Hamann, L., Gogolla, M.: Transformation of UML and OCL Models into Filmstrip Models. In: Ruscio, D.D., Varró, D. (eds.) Proc. 7th Int. Conf. Model Transformation (ICMT 2014). pp. 170–185. Springer, LNCS 8568 (2014)
15. Hilken, F., Niemann, P., Gogolla, M., Wille, R.: Filmstripping and Unrolling: A Comparison of Verification Approaches for UML and OCL Behavioral Models. In: Seidl, M., Tillmann, N. (eds.) Proc. 8th Int. Conf. Tests and Proofs (TAP 2014). pp. 99–116. Springer, LNCS 8570 (2014)
16. Hilken, F., Niemann, P., Gogolla, M., Wille, R.: Towards a Catalog of Structural and Behavioral Verification Tasks for UML/OCL Models. In: Oberweis, A., Reussner, R. (eds.) Proc. Modellierung (MODELLIERUNG'2016). pp. 115–122. GI, LNI 254 (2016)
17. Homik, M., Meier, A.: Designing a GUI for proofs - evaluation of an HCI experiment. CoRR abs/0903.3926 (2009)
18. Lam, V.S.W.: A Formalism for Reasoning about UML Activity Diagrams. Nordic Journal of Comp. 14(1), 43–64 (2007)
19. Lapets, A., Kfoury, A.J.: A user-friendly interface for a lightweight verification system. Electr. Notes Theor. Comput. Sci. 285, 29–41 (2012)
20. Lüth, C.: User interfaces for theorem provers: Necessary nuisance or unexplored potential? ECEASST 23 (2009)
21. Moisuc, D., Revol, S., Snook, C.F.: UML user interface to a proof-based hardware design flow. In: Forum on specification and Design Languages, FDL 2006,. pp. 337–344. ECSI (2006)
22. Pescador, A., Garmendia, A., Guerra, E., Cuadrado, J.S., de Lara, J.: Pattern-based development of domain-specific modelling languages. In: 18th ACM/IEEE MoDELS 2015. pp. 166–175 (2015)
23. Remenska, D., Willemse, T.A.C., Templon, J., Verstoep, K., Bal, H.E.: Property Specification Made Easy: Harnessing the Power of Model Checking in UML Designs. In: FORTE 2014. pp. 17–32 (2014)
24. Salay, R., Chechik, M.: A generalized formal framework for partial modeling. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering - 18th Int. Conf., FASE 2015. Proceedings. LNCS, vol. 9033, pp. 133–148. Springer (2015)
25. Salay, R., Chechik, M., Famelis, M., Gorzny, J.: A methodology for verifying refinements of partial models. Journal of Object Technology 14(3), 3:1–31 (2015)
26. Soeken, M., Wille, R., Drechsler, R.: Verifying Dynamic Aspects of UML Models. In: Design, Automation and Test in Europe, DATE 2011. pp. 1077–1082. IEEE (2011)
27. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL Models Using Boolean Satisfiability. In: Design, Automation and Test in Europe, DATE 2010. pp. 1341–1344. IEEE (2010)
28. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: ECMFA. pp. 69–84 (2011)
29. Wang, X., Rutle, A., Lamo, Y.: Towards user-friendly and efficient analysis with alloy. In: Model-Driven Engineering, Verification and Validation, MoDeVVa@MoDELS 2015. pp. 28–37 (2015)
30. Zloof, M.M.: QBE/OBE: A language for office and business automation. IEEE Computer 14(5), 13–22 (1981)