Towards a Catalog of Structural and Behavioral Verification Tasks for UML/OCL Models

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

Abstract: Verification tasks for UML and OCL models can be classified into structural and behavioral tasks. For both task categories a variety of partly automatic solving approaches exist. But up to now, different interpretations of central notions as, for example, 'consistency' or 'reachability' can be found in current approaches and tools. This paper is designed to clarify central verification notions and to establish a collection of typical verification tasks that are common to multiple approaches and tools. In addition, the verification tasks are categorized with the aim of creating a central catalog of tasks, providing a common understanding of the terms used in model verifications.

1 Introduction

The increasing usage of modelling languages like the *Unified Modelling Language* (UML) and the *Systems Modeling Language* (SysML) and their formalizations have lead to a variety of verification engines for various model descriptions. Along with these tools, a heap of verification tasks were created and defined, each approach with their own definitions. This process has lead to model verification terms, such as, *consistency* or *reachability*, that are used multiple times with differing semantics [CCR08].

In order to establish a general terminology and create a common understanding, this paper takes frequently used verification tasks, describes their goals and categorizes them into a catalog. The categories give a general idea and quick overview of the goals of the tasks assigned to them. In addition, these categories and tasks are divided into structural and behavior topics. The descriptions shall clarify the interpretation of verification tasks. Similar, distinct tasks were given concrete names and descriptions to seperate their overlap. For example, the *consistency* task was split into a *weak* and a *strong* consistency.

The goal of the catalog is a common understanding of the various existing verification tasks to reduce misinterpretations and establish a foundation for communicating about them with a clear understanding of their semantics. The catalog provided in this paper is not meant as a final product, but rather a basis to discuss and extend it.

The remainder of this paper is structured as follows: Section 2 pictures the state of the art and further motivates the categorization of verification tasks. Section 3 introduces a short running example that is used to exemplify the goal of selected verification tasks. Section 4

[1] University of Bremen, Computer Science Department, D-28359 Bremen, Germany
Email: `{fhilken|pniemann|gogolla}@informatik.uni-bremen.de`

[2] Johannes Kepler University, Computer Science Department, A-4040 Linz, Austria
Email: `robert.wille@jku.at`

first defines a metamodel to represent verification tasks and then finished with the actual catalog, categorizing and describing the verification tasks. Section 5 wraps up the paper with a conclusion.

2 Motivation

Modeling languages such as the *Unified Modeling Language* (UML) or the *Systems Modeling Language* (SysML) together with textual constraints, e.g., provided by the *Object Constraint Language*, have been established to specify the design of complex systems. They provide different concepts such as class diagrams, sequence diagrams, or activity diagrams which are expressive enough to formally specify a complex system, but hide specific implementation details. Since modeling languages permit formal descriptions, they additionally enable the verification of the respective specification already in the absence of a specific implementation[1].

The corresponding verification tasks can be divided into

- *Structural Verification Tasks*, where a single system state is considered, as well as
- *Behavioral Verification Tasks*, where a sequence of system states as well as their transitions (e.g., described by operations with pre- and postconditions) is considered.

For both categories of verification tasks, a variety of (automatic) solving approaches have been introduced in the recent past [CCR08, SWD11, An07, Ba12, CKZ11, EW04, La07, Ro14]. However, until today different interpretations and terminologies exist for the respectively considered verification tasks.

For structural verification tasks, definitions as proposed e.g. in [GKH09] became rather established. Nonetheless, even in this context, multiple notions and variations can be found in the literature. For instance, consider the well-established task of checking “consistency”, i.e. investigating whether a model description is consistent in that sense that an instantiation of the model exists which satisfies all of the model’s constraints: in [CCR08], any non-empty instantiation of the model is accepted, while [GHH14] forces each class of the model to be instantiated at least once.

For behavioral verification tasks, so far a comprehensive list of tasks has not even been attempted at all (to the best of our knowledge). In contrast, for other areas of validation and verification in modeling, similar compilations of verification tasks and techniques have been presented, e.g. a survey on tasks for model transformations in [CS13] or a survey on modeling techniques for behavioral verification of software product lines in [Be15].

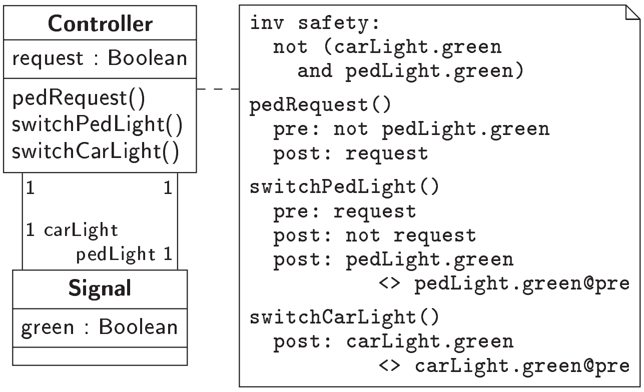
In this work, we aim for providing a unique and clear definition of important verification tasks that can be applied on UML/OCL models. This includes a comprehensive consideration of both, structural and behavioral issues. Thus far, verification tasks are often

[1] In model-driven engineering, it is common to apply so called model transformations to automatically transform models into a different description mean or language during the design process. In this context, it is an important task to validate/verify whether source and target model of a transformation are equivalent. However, this is out of the scope of the present paper where we focus on the verification of stand-alone model descriptions

referenced using different terms or using the same term, but having in mind different meanings. By presenting a fine granular differentiation of tasks in the following sections, we try to reduce misinterpretations and establish a common basis for an improved and clarified communication about verification tasks.

3 Traffic Light Running Example

In this section we introduce a simplified pedestrian traffic light preemption which will serve as a running example to illustrate concepts discussed in the next section. The corresponding model is depicted in Fig. 1. The main class of the example is the `Controller` which is connected to exactly two traffic light signals, one for cars (`carLight`) and one for pedestrians (`pedLight`). For simplicity, we assume two-state signals (green light on/off). With the operation `pedRequest()`, pedestrians express the desire to cross the road from either side. The controller stores these requests in the `request` attribute and switches the corresponding signals using the `switchPedLight()` and `switchCarLight()` operations. To prevent accidents, the invariant `safety` ensures that pedestrians and cars may not both face a green light (indicating a safe crossing) at the same time.



```
inv safety:
  not (carLight.green
    and pedLight.green)
pedRequest()
  pre: not pedLight.green
  post: request
switchPedLight()
  pre: request
  post: not request
  post: pedLight.green
           <> pedLight.green@pre
switchCarLight()
  post: carLight.green
           <> carLight.green@pre
```

Fig. 1: Traffic light running example.

4 Categorizing Verification Tasks

We have identified a variety of verification tasks that are used in model checkers and extracted use cases from them. These use cases were assigned to five basic categories, giving a quick overview of the general goal of each task. The five categories are *Consistency*, *Independence*, *Reachability*, *Executability* and *Consequence*. The *Consistency* category represents general instantiability use cases, the *Independence* category describes use cases checking relations of model elements, the *Reachability* category contains use cases that check if certain goals are reachable when the behavior of the model is simulated, the *Executability* category specifies use cases that examine possible transitions between system states and, finally, the *Consequence* category characterizes use cases that deduct model properties and put model elements into relation. These categories naturally divide into two areas: *structural* and *behavioral* tasks.

Figure 2 illustrates the extracted use cases and relations in between them and the general categories. The dotted line in the middle indicates the separation between structural and behavioral verification tasks, with structural tasks on the left of the line and behavioral tasks on the right, respectively. As for the five categories, the classification into structural and behavioral verification tasks is not as strict, as behavioral tasks may be categorized

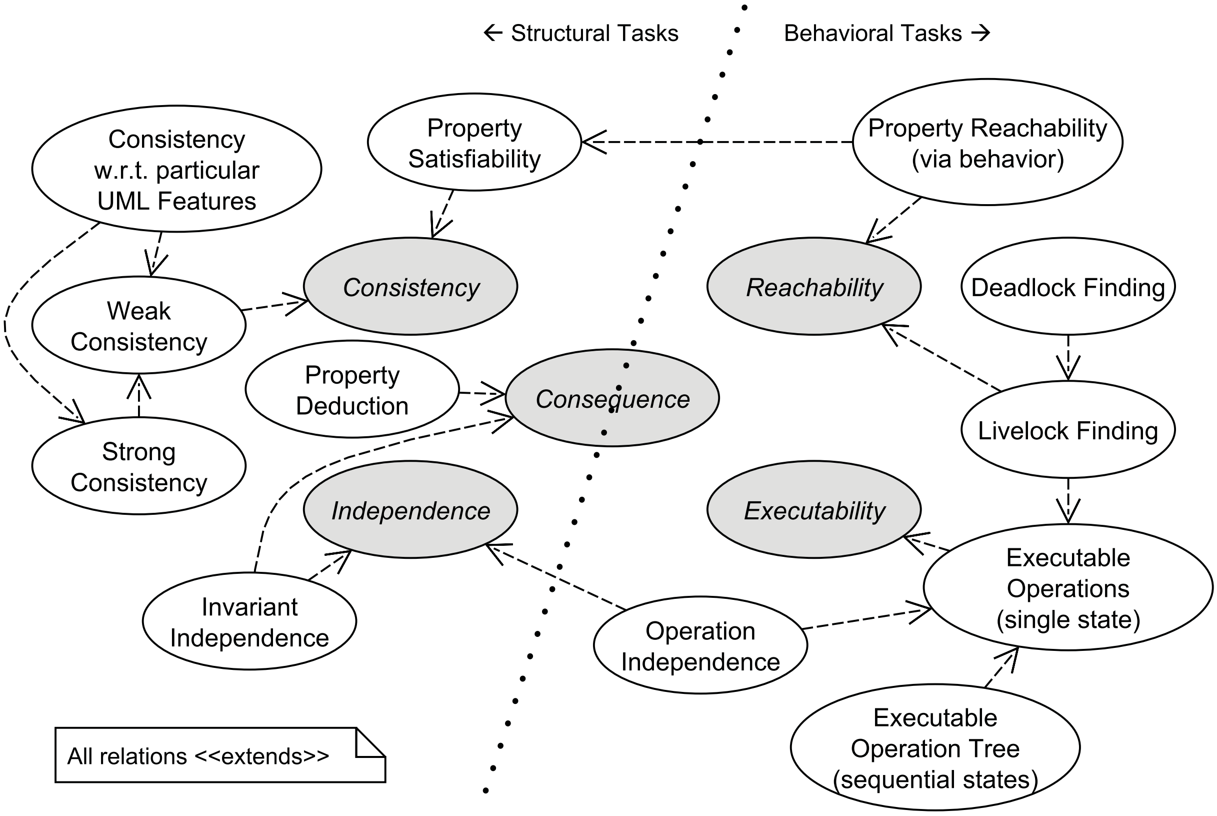


Fig. 2: Overview of verification task catalog.

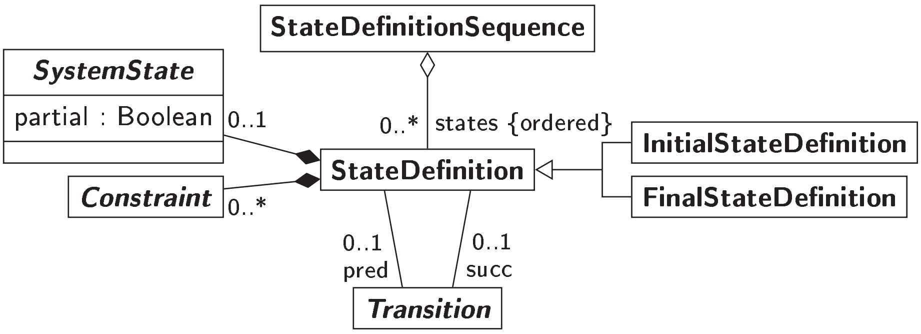


Fig. 3: Data model of inputs for verification tasks.

in a structural category (*Operation Independence*) or extend on structural tasks (*Property Reachability*). The listed verification tasks will be explained later in this section.

4.1 Verification Task Metamodel

In order to describe verification tasks in a formal fashion, we use the metamodel shown in Fig. 3. It shows an abstract metamodel that we use as a baseline for the declaration of the verification task input. The metamodel describes a structure that contains the information verification engines need to solve a certain verification task. The model creates a skeleton of information that must be filled in with a valid assignment by the verification engine.

Structural tasks utilize the `StateDefinition`, in the center of Fig. 3. This class represents a single abstract system state, which imposes no restrictions on a verification engine when generating a result. Using the abstract class `SystemState`, a concrete assignment for this state can be given. The attribute `partial` determines, whether more elements may be added to this system state or not. Finally, the class `Constraint` allows to add (boolean) properties to a `StateDefinition` that have to be satisfied in the result. These methods to describe a system state can be mixed as necessary. To give a concrete example, a `StateDefinition` might be an object diagram or a state in a state machine.

While structural tasks only need the three classes mentioned above, behavioral tasks have access to additional information about the sequence in which these defined states occur (`StateDefinitionSequence`), the predecessors and successors of the states as well as the order and possibly the type of transitions between them. For example, the abstract class `Transition` might be extended to represent operation calls, state transitions in a state machine or signals. Finally, a `StateDefinition` can explicitly be declared as the initial or final state.

Figure 4 shows an example for a *Reachability* verification task pictured as an instance of the metamodel[2]. In the example, an initial and a final state is given by object diagrams. In the final state, both signals are set to green and the task is to find valid transitions from the initial to the final state, using the behavior defined in the model from Sect. 3. Since there is no path of transitions given in between the two system states, the amount and type of transitions is not restricted. Additionally, the system states could be further restricted by constraints in which the objects can be accessed using their names. The object names are also used to map them in between system states.

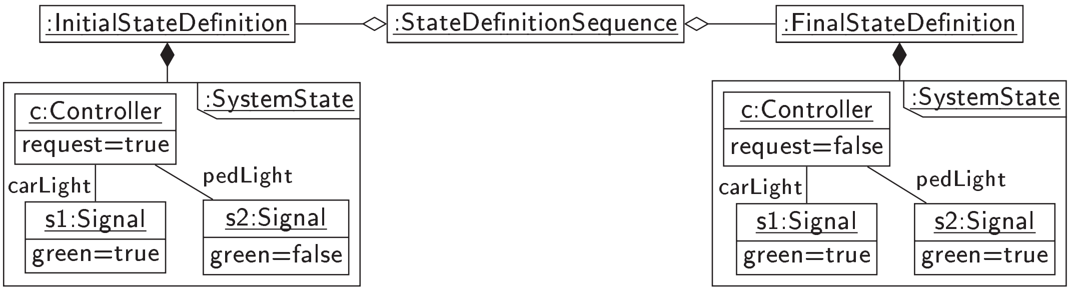


Fig. 4: Verification task metamodel example to define a reachability task with object diagrams.

4.2 Verification Tasks

In the following, the categories and their associated verification tasks from Fig. 2 are detailed and examples are given to illustrate the goals of selected tasks. The list of verification tasks, as framed in this work, is not meant to be complete. However, the provided list is a good viewpoint to show verification tasks that model checkers should be able to perform on UML/OCL models. We encourage others to extend the list and iteratively collect a more complete catalog of verification tasks.

4.2.1 Consistency

A *Consistency* verification task describes the instantiability property of a model, taking into account different sets of constraints applied, e.g., explicit and implicit model constraints, additional properties that serve as a certain verification goal, or even a reduced

[2] Due to space restrictions, we leave out exact details, how the abstract classes are extended to represent the information as an object diagram.

set of constraints. This category contains crucial verification tasks like showing whether a model contains contradictions and, therefore, might not be instantiable at all. Consistency problems are structural problems and do not involve behavior, like the execution of operations.

Weak Consistency This task describes the general instantiability of a model. The goal is to generate a system state that uses at least some model elements while satisfying all model constraints. Note that invariants assigned to classes that are not instantiated, are satisfied by design in the standards.

Strong Consistency This task is an extension of *Weak Consistency* by the property that *all* model elements have to be considered in the generated system state, i.e., at least one object of all classes and one link of all associations have to be instantiated.

Consistency w.r.t. particular UML Features This task extends the former two tasks allowing particular UML features, such as multiplicities, aggregation and composition rules or invariants, to be ignored.

Property Satisfiability This task represents a consistency task including external properties in addition to the model. The goal is to find a valid system state satisfying all model constraints plus the additional properties. Figure 5 shows a small example requiring at least a system state with a controller `c`. In addition, a specific property is specified as an OCL constraint, requiring both signals of this controller show the green signal, which fails due to the invariant `safety`.

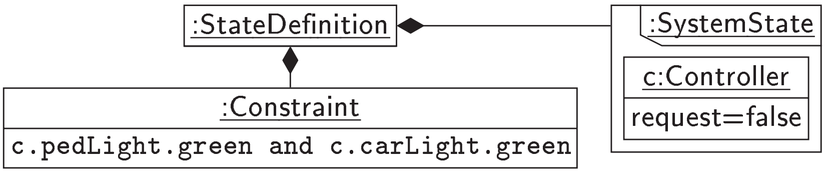


Fig. 5: Verification task model for property satisfiability task.

4.2.2 Independence

Independence describes verification tasks that reason about (in)dependencies between model elements. This includes any type of dependencies that can exist between, e.g., attributes, roles or invariants. In addition, tasks setting these dependencies in relation also belong in this category.

Invariant Independency The goal of this task is to check whether invariants exist that are implicitly specified by one or more others and are therefore always satisfied when the dependant invariants are satisfied. Further extensions of this task is the identification of which invariants imply the dependent invariant.

4.2.3 Consequence

Verification tasks in the category *Consequence* describe tasks that deduct information from a model. These consequences are inherent in the model and are given by the model constraints, e.g., multiplicities, invariants or more complex deductions.

Property Deduction This task describes the action of identifying information not explicitly in a model, but that are clearly implied by one or more model elements from the model. In contrast to the *Property Satisfiability* task, these information deducted from the model are not restricted to boolean properties.

4.2.4 Reachability

Reachability verification tasks include all tasks with a certain defined goal in mind that is reached by executing the behavior of a model such as operations or state machine state transitions. In contrast to *Consistency* verification tasks, *Reachability* tasks involve at least two system states that are connected by model transitions, defined by the behavior of the model.

Property Reachability Similar to the *Property Satisfiability* verification task, this task checks the satisfiability of properties in a model. This tasks, however, tries to satisfy them by executing model transitions and additionally allows to specify initial and intermediate system states that must be included in the simulation. The properties to be reached can be given as system states or constraints, as defined in the metamodel in Fig. 3. In the running example, it is desirable to reach a state where the pedestrians are finally allowed to cross the street, when the signals are currently allowing the cars to cross, and vice versa.

4.2.5 Executability

In the *Executability* category are all verification tasks that focus on the transitions and their contracts between system states. While this paper focuses on operation calls, the metamodel in Sect. 4.1 allows for any form of state transition.

Livelock Finding This task identifies state transitions that result in a *livelock*, i.e., the system is in a state where there is no possible sequence of transitions to reach a defined end state, while transitions can still be executed.

Deadlock Finding This verification task is the extension of the *Livelock* task, searching for reachable system states, where the system comes to a complete halt and no further transition is possible, without being in a defined end state. This task can make sure that, in the running example, there is no possibility to get into a state where only either cars or pedestrians are allowed to cross the street (forever).

Executable Operations The goal of this verification task is to identify all executable transitions of a single system state. In the running example this is achieved by evaluating the preconditions of all operations against the given system state. This task is the basis for many other verification tasks.

Executable Operation Tree This verification task extends the previous task by checking the possible transitions not only for a single system state, but also simulating the operation calls and iteratively evaluate the executable state transitions, building a full tree of operation call sequences up to a given depth. Again the task can be restricted by giving a certain final state as a goal or constraining the transition sequence using simple OCL. The constraints on the sequence can be as complex as temporal logic.

Operation Independence This task introduces the detection of dependencies in operations from the former verification task. An example is the identification of mutually exclusive operation calls, i.e., identifying operation calls that are never available at the same time in a single system state.

5 Conclusion

We have presented a catalog of general verification tasks for UML/OCL verification tasks including their categorization into five groups. All tasks are individually detailed to establish a fine granular differentiation between their goals. These definitions shall help modellers to communicate with each other and unify the term usage in verification engines. In addition, we have presented a metamodel to represent these verification tasks in a formal fashion.

References

[An07] Anastasakis, Kyriakos; Bordbar, Behzad; Georg, Geri; Ray, Indrakshi: UML2Alloy: A Challenging Model Transformation. In: MoDELS. Springer, pp. 436–450, 2007.

[Ba12] Banerjee, Ansuman; Ray, Sayak; Dasgupta, Pallab; Chakrabarti, P. P.; Ramesh, S.; Ganesan, P. Vignesh V.: A dynamic assertion-based verification platform for validation of UML designs. ACM SIGSOFT Software Engineering Notes, 37(1):1–14, 2012.

[Be15] Benduhn, Fabian; Thüm, Thomas; Lochau, Malte; Leich, Thomas; Saake, Gunter: A Survey on Modeling Techniques for Formal Behavioral Verification of Software Product Lines. In: Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems. VaMoS '15. ACM, pp. 80:80–80:87, 2015.

[CCR08] Cabot, Jordi; Clarisó, Robert; Riera, Daniel: Verification of UML/OCL Class Diagrams using Constraint Programming. In: First International Conference on Software Testing Verification and Validation, ICST 2008. IEEE Computer Society, pp. 73–80, 2008.

[CKZ11] Choppy, Christine; Klai, Kais; Zidani, Hacene: Formal Verification of UML State Diagrams: A Petri Net based Approach. Softw. Eng. Notes, 36(1):1–8, 2011.

[CS13] Calegari, Daniel; Szasz, Nora: Verification of Model Transformations: A Survey of the State-of-the-Art. Electronic Notes in Theoretical Computer Science, 292:5 – 25, 2013. Proceedings of the XXXVIII Latin American Conference in Informatics (CLEI).

[EW04] Eshuis, Rik; Wieringa, Roel: Tool Support for Verifying UML Activity Diagrams. ITSE, 30(7):437–447, 2004.

[GHH14] Gogolla, Martin; Hamann, Lars; Hilken, Frank: Checking Transformation Model Properties with a UML and OCL Model Validator. In (Amrani, Moussa; Syriani, Eugene; Wimmer, Manuel, eds): Proc. 3rd Int. Workshop on Verification of Model Transformation (VOLT'2014). CEUR Proceedings, Vol. 1325, pp. 16–25, 2014.

[GKH09] Gogolla, Martin; Kuhlmann, Mirco; Hamann, Lars: Consistency, Independence and Consequences in UML and OCL Models. In (Dubois, Catherine, ed.): Tests and Proofs, Third International Conference, TAP. volume 5668 of LNCS. Springer, pp. 90–104, 2009.

[La07] Lam, Vitus S. W.: A Formalism for Reasoning about UML Activity Diagrams. Nordic Jrnl. of Comp., 14(1):43–64, 2007.

[Ro14] Rodríguez, Ricardo J.; Fredlund, Lars-Åke; Herranz-Nieva, Ángel; Mariño, Julio: Execution and Verification of UML State Machines with Erlang. In: Software Engineering and Formal Methods. pp. 284–289, 2014.

[SWD11] Soeken, Mathias; Wille, Robert; Drechsler, Rolf: Verifying Dynamic Aspects of UML Models. In: DATE. IEEE, pp. 1077–1082, 2011.