Towards a Base Model for UML and OCL Verification[⋆]

Frank Hilken, Philipp Niemann, Robert Wille, and Martin Gogolla

University of Bremen, Computer Science Department
D-28359 Bremen, Germany
{fhilken,pniemann,rwille,gogolla}@informatik.uni-bremen.de

Abstract. Modelling languages such as UML and OCL are more and more used in early stages of system design. These languages offer a huge set of constructs. As a consequence, existing verification engines only support a restricted subset of them. In this work, we propose an approach using model transformations to unify different description means within a so called base model. In the course of this transformation, complex language constructs are expressed with a small subset of so-called core elements. This simplification enables to interface with a wide range of verification engines with complementary strengths and weaknesses. Our aim is that, guided by a structural analysis of the base model, the developer can choose the most promising verification engine.

1 Introduction

In recent years Model-Driven Engineering (MDE), has become more and more important. In this context, the Unified Modelling Language (UML) and the Object Constraint Language (OCL) are de facto standards to describe systems and their behaviour. Identifying errors early in the design of such systems using validation and verification techniques is an important task, though finding the right verification approach is not trivial, since most approaches concentrate on only one UML diagram type and restrict the set of supported diagram types and language constructs.

In this paper, we propose the idea of a so-called *base model* that, using model transformations, combines the information of several UML diagram types into a single diagram and reduces the set of language constructs to a minimum. The language constructs are split into three categories: (1) core elements that are directly used in the base model; (2) transformed elements that are represented in the base model using only core elements, reducing the used amount of language constructs; and (3) unsupported elements that are excluded because of their low relevance or because of their infeasibility in the context of verification. The base model can be seen as a substantially reduced version of UML with a strong focus on compatibility to verification engines.

⋆ This work was partially funded by the German Research Foundation (DFG) under grants GO 454/19-1 and WI 3401/5-1 as well as within the Reinhart Koselleck project DR 287/23-1.

On the basis of the base model, we perform a structural analysis separated into several categories in order to identify the most promising verification engine for the particular model under development. Once a verification engine is chosen, further model transformations can remove remaining incompatibilities in the same manner as from the source model to the base model. All previously mentioned transformations are performed on the UML and OCL layer in order to have a unified process for every verification engine.

The structure of the paper is as follows. In Sect. 2, the base model is discussed including the process flow, definitions and transformations. Section 3 describes the analysis of the base model and how a solver is chosen. Lastly, the transformations for the solving engine are shown in Sect. 4. In Sect. 5, we discuss related work before concluding the paper in Sect. 6.

2 A Base Model for UML and OCL Verification

The base model provides an interface between arbitrary UML/OCL model descriptions and validation and verification tools. The goal of the base model is to represent a unified basis for model descriptions combining various UML diagram types while retaining a maximum compatibility to the original model and to solvers.

Figure 1 shows the process flow when using such a base model. The source model description consists of various UML diagrams enhanced with OCL expressions in order to specify the system and its behaviour. All diagrams conjoined are transformed and combined into a base model. Using the base model, a structural analysis can provide hints for an appropriate solver selection. Once the solver is chosen, the base model is transformed into a solver-specific base model elimi-

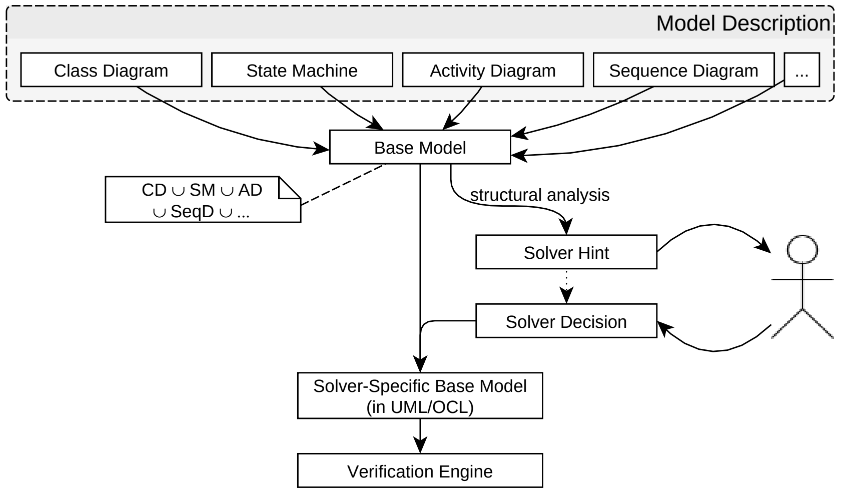


Fig. 1. Process flow employing the base model

nating modelling constructs not explicitly supported by the solver and replacing them with simpler representations. All transformations are automatic and do not require manual interaction unless the user wants to choose a specific solving engine. Finally, the solver specific base model serves as input for the verification engine.

We define the basis of the source model description to be a class diagram optionally enhanced with operations specifying model behaviour. Other diagram types, such as state machines, activity diagrams and sequence diagrams, can further define the system and its behaviour. The result of the model description transformation is a class diagram with optionally employed operations using pre- and postconditions to describe the system's behaviour. The information of the various diagram types is transformed into either class invariants or pre- and postconditions of the operations. Additionally, the base model only features a reduced set of UML and OCL features, the core elements. Elements that are not part of these are transformed into simpler representations using only core elements to increase compatibility to solving engines.

Example 1. For instance, an aggregation from the source model is transformed into an association plus an invariant, demonstrated in Fig. 2 with a simple mother child relationship.

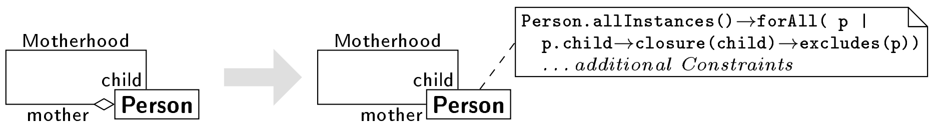

Motherhood
child
mother
Person
Motherhood
child
mother
Person
Person.allInstances()→forAll(p |
p.child→closure(child)→excludes(p))
... additional Constraints

Fig. 2. Transformation of an aggregation into association plus constraint

A structural analysis on the base model is now able to determine the relevance of certain criteria and can give hints about which solving engines are most effective. This is described in more detail in Section 3. A developer can combine these hints and their knowledge about the system to choose the most appropriate solver. Alternatively, if the user interaction is not desired, the solving engine that the analysis has determined to be best can be chosen directly. In the case that the solving engine does not support all features present in the base model, these elements are further transformed into solver compatible elements, resulting in the solver specific base model. Most likely, these transformations are required to enable the usage of such verification engine anyway and therefore does not influence the scalability. Additionally, note that all transformations are performed on the UML and OCL layer and only have to be implemented once instead of individually for each solving engine.

Table 1. UML elements in the base model (*symbols:* ✓ core element; ○ transformed element (using only core elements); × unsupported element)

Class features	**Association features**	**Operation features**
✓ Class	✓ Binary Association	✓ Operation (non query)
○ Abstract Class	○ N-ary Association	✓ Parameter
○ Inheritance	○ Aggregation	× Return Value
○ Multiple Inheritance	○ Composition	✓ Pre-/Postcondition
✓ Attribute	✓ Multiplicity	× Nested Operation Call
○ Initial Value	○ Association Class	○ Query Operation
○ Derived Value	○ Qualified Association	✓ Parameter
✓ Enumeration	× Redefines, Subsets, Union	✓ Return Value
✓ Invariant		× Recursion

2.1 Elements of the Base Model

To create a unified model representing the input model description with a maximum compatibility to verification engines, the base model has a reduced feature set of UML and OCL elements. Table 1 shows the supported UML elements in the base model and marks those elements that are transformed into a more basic representation. The supported elements form a basis to represent most UML features either directly or using the available elements plus invariants and are supported by the majority of solving engines. Many transformations of complex model elements into the core elements are described in [7].

The essential elements of the base model are defined by the class and association features. The core elements, representing the "atoms" of the base model, are *enumerations*, *classes* with their *attributes* and *invariants*, and *binary associations* with their *multiplicities*. Most of these elements cannot be represented by simpler description means. Other structural features, such as *association classes* and *aggregations*, are transformed into representations using the core elements. Very specific features, like *redefines* and *subsets*, are not supported for now.

The operation features define the behaviour of the base model using *operations* with optional *parameters* and *pre-* and *postconditions*. *Return values* are not supported until *nested operation calls* are, which we can not handle currently. *Query operations* are integrated into the expression they are used in, which is also the reason why we do not support *recursion* for these.

As for OCL, we support a basic set of features to cover the majority of OCL expressions. The data types `Boolean` and `Integer` are fully supported. `Strings` are only represented by an ID, allowing for a comparison on an instance level, but not on a character level. This also excludes string operations like `concat`. We also support the OCL collection types `Set(T)`, `Bag(T)`, `Sequence(T)` and `OrderedSet(T)` with the essential collection operations, including but not limited to: constructors and manipulation operations `including` and `excluding`; membership tests `includes` and `excludes`; quantifiers `forAll` and `exists`; count operations `size`, `isEmpty` and `notEmpty`; filters `select` and `reject`; and `closure`.

The advantages of the transformation at this early stage are numerous. First and foremost, the solving engines do not require an encoding for the different UML elements. Instead, only the base elements have to be supported. The implementation of the transformation of the complex elements is only required once on the UML and OCL layer. Therefore, it is easier to provide support for more solving engines and they are easier interchangeable. There might be cases, in which a solving engine has a better encoding for a feature than the transformation in the base model, but we expect the transformed elements (marked with a $\circ$ symbol) to be affected least. Also, the assignment of elements to their respective category may change if case studies expose an advantage.

Secondly, the unification provides a solid baseline for the structural analysis to give optimal hints. The unification also helps to break down the model into the relevant categories like quantifiers and special OCL constructs, such as the `closure` expression.

Reasons, why certain elements are not part of the base model at all, are their incompatibility to be represented within the base elements and their high complexity which is not supported by many solving engines, e.g. nested operation calls. Also, elements depending on other unsupported elements are not part of the base model, e.g. return values for non query operations which are only useful if an operation invokes another operation to work with its return value. Note the difference between the transformation of UML and OCL features in the base model and the elimination of UML and OCL features in the solver specific base model, whose sole reason is the compatibility to the solving engine.

2.2 Incorporation of Various Diagram Types

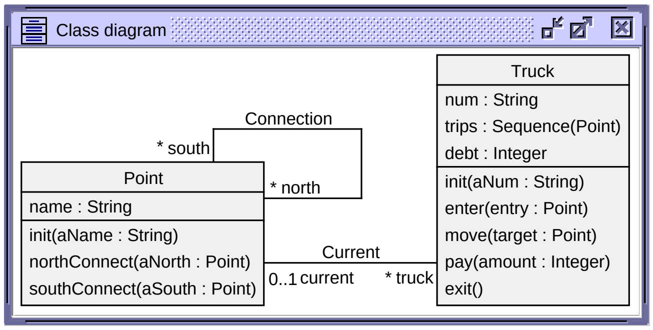
The various diagram types of UML provide different information about the model. A class diagram defines the data structure and operations of a system. State machines, for example, are able to further restrict operation invocations in addition to the operation preconditions. They can also define additional effects of operations as well as state invariants for object states. Activity diagrams provide operation behaviour in form of an implementation. Sequence diagrams again restrict the operation execution order on a different layer than state machines.

Most verification engines specialize on one of these diagram types or require a specific combination of them to be able to accept the system definition. We want to combine the information of the various diagram types into one base model using model transformations, accumulating all information. This way, the requirements for the verification engine are kept minimal and the selection of solvers increases.

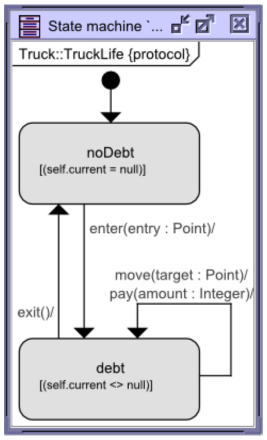
In a first step, we plan to incorporate the information of protocol state machines – without events – into the base model, thereby combining class diagrams with operations and state machines.

Example 2. An example model specifying a Toll Collect[1] system is defined in Fig. 3. It consists of a class diagram (Fig. 3a) and a state machine (Fig. 3b).

[1] www.toll-collect.de/en/home.html



(a) Class diagram with operations



(b) State machine

Fig. 3. Toll Collect example system

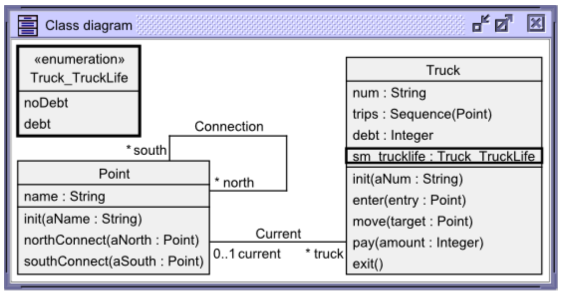
The system features a network of points on which trucks can enter and move around. For each visited point the debt of a truck increases. To leave the network, a truck has to pay its debt. The state machine ensures that the truck can only enter the network when it is not currently in it and only move around and exit the network when it is inside. The state invariants of the state machine further enforce these properties.

To transform the state machine into the class diagram, the states are represented as an enumeration. An additional attribute `sm_trucklife` is added to the corresponding class to save the current state. These changes are illustrated and highlighted in the class diagram in Fig. 4. Using the enumeration attribute, the state invariants can be represented as class invariants that only trigger if the object is in the required state. Constraints given by the transitions of the state machine, including guards and postconditions, are transformed into pre- and postconditions of the operation definitions in the class diagram, as shown on the right in Fig. 4.

3 Solver Selection

So far, various approaches for the verification of UML and OCL models have been presented [2,4,13]. The main idea of these approaches is to encode verification problems in a language that can be passed to a dedicated solving engine and interpret the results at the level of UML and OCL.

In this context, a large variety of languages and solving engines has been suggested. Approaches using *theorem provers* like Isabelle [2], reformulating the problem as a *Constraint Satisfaction Problem (CSP)* [4], using intermediate languages like *Alloy* or *Kodkod* [1,13] though finally resulting in an instance of



```
-- state invariant noDebt
context Truck inv:
  sm_trucklife = #noDebt
  implies current = null

-- op. Truck::exit()
pre: sm_trucklife = #debt
post: sm_trucklife =
    #noDebt
```

Fig. 4. State machine information integrated into class diagram

Boolean Satisfiability (SAT), or using a direct encoding in the more general language of *Satisfiability Modulo Theories (SMT)* [12] have been proposed.

All these solving engines rely on different abstractions and accordingly employ complementary solving schemes. Hence, it is important to choose an appropriate solver for a given UML and OCL description and verification problem. For instance, Kodkod and Alloy have been designed for problems of relational logic. They support set theory including transitive closure, and are, thus, especially suitable for OCL constraints expressing relations between model elements. In contrast, SMT offers theories for the efficient handling of bit-vectors (Integers) and corresponding arithmetic operations, while CSP particularly supports abstract data types (e.g., collections/lists). Beyond that, there are different approaches to cope with quantifiers (existential and universal) like *Quantified Boolean Formulas (QBF)* [6] or integration of quantifiers into bit-vector logic in the SMT-solver Z3 [5].

In order to benefit from these particular, complementary strengths, we suggest a structural analysis of the model, especially of the OCL expressions, with respect to collection types (e.g., `Set(T)` and `Sequence(T)`), arithmetic components ($+$, $-$, $*$, $/$), relational components (e.g., `size()` and `closure()`) and quantifiers (`forAll` and `exists`). This analysis shall provide hints which solver might be most adequate for the verification of the given base model. As adequateness can hardly be quantified in terms of absolute scores, the results shall be presented to the developer in an interactive procedure. First, the above criteria are listed together with their relevance for the given model (high, moderate, low or none) as determined by the structural analysis. These values may then be adjusted by the developer in order to incorporate her own rating. In a second step, a recommendation is given which solvers are most appropriate for the given problem based on how their strengths and weaknesses fit to the profile of relevance derived in the first step. This feedback shall also contain information about model elements that are not directly supported by the solvers (e.g. Reals, Strings, or certain OCL constructs) and will be ignored or transformed into simpler means. Assisted by this advice, the developer can finally decide on which solving paradigm might be most appropriate to use. Alternatively, the developer may also let this decision be made by the framework automatically.

4 Solver-Specific Base Model

Though many different solving technologies are available, basically all approaches do not cover the complete OCL language. They rather restrict themselves to a subset for which an encoding exists, i.e. a model transformation to the solver level. Consequently, before passing the base model to the chosen solver, unsupported model elements have to be transformed.

Example 3. Consider the collection type `Sequence(Point)`, as contained in the Toll Collect example model from earlier (c.f. Fig. 3a). If the chosen solver technology does support sequences (e.g. Kodkod, Alloy, or CSP) no transformation is necessary at this step and the base model is passed to the verification engine unchanged. However, if the chosen solver does not support sequences, these are to be transformed. This can be done by introducing two new classes `PointSequence` and `PointSequenceElement`, organized as a linked list with references to the beginning and the end of the sequence, shown in Fig. 5. The resulting solver-specific base model is then passed to the verification engine and finally to the solver.

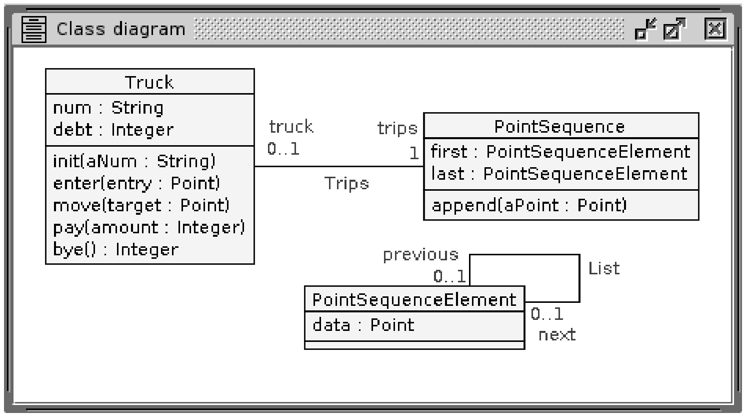


Fig. 5. Replacing OCL collection type *Sequence* by introducing a new UML class

UML and OCL are rich languages with many facets which makes it hard to (1) determine an encoding for each component (e.g. the encoding of collections in SMT [11]) and (2) requires a large effort to realize these encodings in (prototypic) implementations. Addressing this issue, the transformation to a solver-specific base model can also help to lower the threshold for incorporating new solving engines. More precisely, by providing a *whitelist* of supported UML and OCL components, the range of tractable model elements can be enlarged by OCL transformations (e.g. using `select` to express `reject` or using `COL`$\rightarrow$`size()=0` for `COL`$\rightarrow$`isEmpty()`) or by more substantial, structural modifications (e.g. transforming the collection type `Sequence(T)` to separate classes).

Though these transformations are performed automatically, the developer shall be able to access which kind of transformation is performed. This allows for a judgement whether they may interfere with the addressed verification task or whether they are applied to secondary components of the model only.

5 Related Work

There are several contributions that can be related to our present work. The fundamental idea of a base model within a generic verification methodology for system descriptions expressed in terms of UML and OCL has been presented in [14]. Verification of other description means than class diagrams has been addressed by many approaches, e.g. [8], while using OCL in order to express complex UML class diagram properties by simpler means, has been discussed in [7]. Validation and verification of such model transformations, e.g. using the ATLAS Transformation language (ATL) [3], is an active field of research. A comparison of such verification techniques has been presented in [9]. Also to the solver end, a similar, but more quantitative comparison between different solving paradigms has been conducted [10]. The results indicate a predominance of SMT, especially for large benchmarks. However, this comparison only considers a single class of models of the same type which are only varied in size, and a further comparison is needed for models of other types.

6 Conclusion and Future Work

In this work, we presented a model transformation of heterogeneous model descriptions to a unified base model, which enables to consider more comprehensive descriptions for verification. In the course of this transformation, a small set of core elements is used to express a large and rich set of UML and OCL constructs. In doing so and excluding several language constructs due to their minor relevance or general infeasibility with respect to verification, we expect to improve compatibility to verification engines without significantly restricting model support. We have exemplarily shown the incorporation of state machines into the base model by means of an example. Details of this transformation as well as the transformation of other diagram types are left for future work.

Beyond that, we have identified and suggested categories of modelling constructs that may affect the performance of verification engines, if these are applied to models that contain a considerable amount of those constructs. However, their actual impact on solving times and performance has not been examined thoroughly so far and remains open for further research.

Finally, by providing further transformations of the base model, we are able to obtain solver-specific base models employing only language constructs that are supported by the addressed solver. Instead of leaving the implementation of these transformations to the verification engines, they are performed transparently at a higher level of abstraction. This allows us to interface with a wider range of solvers, potentially at the price of a little overhead and loss of performance.

Overall, the base model framework provides us with a generic interface between heterogeneous model descriptions and verification engines.

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. Brucker, A.D., Wolff, B.: The HOL-OCL book. Tech. Rep. 525, ETH Zurich (2006)
3. Büttner, F., Cabot, J., Gogolla, M.: On validation of ATL transformation rules by transformation models. In: Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation. p. 9. ACM (2011)
4. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on. pp. 73–80. IEEE (2008)
5. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
6. Giunchiglia, E., Narizzano, M., Tacchella, A.: Qube: A system for deciding quantified boolean formulas satisfiability. In: Automated Reasoning, pp. 364–369. Springer (2001)
7. Gogolla, M., Richters, M.: Expressing UML Class Diagrams Properties with OCL. In: Clark, T., Warmer, J. (eds.) Advances in Object Modelling with the OCL, pp. 86–115. Springer, Berlin, LNCS 2263 (2001)
8. Kaufmann, P., Kronegger, M., Pfandler, A., Seidl, M., Widl, M.: Global State Checker: Towards SAT-Based Reachability Analysis of Communicating State Machines. In: Boulanger, F., Famelis, M., Ratiu, D. (eds.) MoDeVVa@MoDELS. CEUR Workshop Proceedings, vol. 1069, pp. 31–40. CEUR-WS.org (2013)
9. Lano, K., Kolahdouz-Rahimi, S., Clark, T.: Comparing verification techniques for model transformations. In: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation. pp. 23–28. ACM (2012)
10. Saadatpanah, P., Famelis, M., Gorzny, J., Robinson, N., Chechik, M., Salay, R.: Comparing the effectiveness of reasoning formalisms for partial models. In: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation. pp. 41–46. ACM (2012)
11. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP. Lecture Notes in Computer Science, vol. 6706, pp. 152–170. Springer (2011)
12. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL Models Using Boolean Satisfiability. In: DATE. pp. 1341–1344. IEEE (2010)
13. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: France, R.B., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA. Lecture Notes in Computer Science, vol. 6698, pp. 69–84. Springer (2011)
14. Wille, R., Gogolla, M., Soeken, M., Kuhlmann, M., Drechsler, R.: Towards a Generic Verification Methodology for System Models. In: Macii, E. (ed.) DATE. pp. 1193–1196. EDA Consortium San Jose, CA, USA / ACM DL (2013)