From UML/OCL to Base Models: Transformation Concepts for Generic Validation and Verification

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

[1] University of Bremen, Computer Science Department, 28359 Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{fhilken,pniemann,gogolla,rwille}@informatik.uni-bremen.de

Abstract. Modeling languages such as UML and OCL find more and more application in the early stages of today's system design. Validation and verification, i.e. checking the correctness of the respective models, gains interest. Since these languages offer various description means and a huge set of constructs, existing approaches for this purpose only support a restricted subset of constructs and often focus on dedicated description means as well as verification tasks. To overcome this, we follow the idea of using model transformations to unify different description means to a *base model*. In the course of these transformation, complex language constructs are expressed by means of a small subset of so-called *core elements* in order to interface with a wide range of verification engines with complementary strengths and weaknesses. In this paper, we provide a detailed introduction of the proposed base model and its core elements as well as corresponding model transformations.

Keywords: model transformation, UML, OCL, metamodel, validation and verification, base model.

1 Introduction

In recent years, *Model-Driven Engineering* (MDE) has become more and more popular, and modeling languages are more and more used in early stages of today's system design. In this context, the *Unified Modeling Language* (UML) and the *Object Constraint Language* (OCL) are de facto standards to describe systems and their behavior. They provide formal descriptions of system models which, besides others, can be applied for purposes of validation and verification. Indeed, identifying flaws and errors early in the design of such systems using validation and verification techniques is an important task. In our work, we focus on automatic (i.e. *push button*) methods which require almost no further knowledge on the underlying verification technique and, thus, can be used by every system designer.

However, developments in the previous years led to an "inflation" of different verification approaches for designs given in terms of modeling languages (this is discussed in detail later in Sect. 2). Finding an appropriate verification approach

is a non-trivial task, since most approaches focus on one particular UML diagram type and additionally restrict the set of supported language constructs. This poses a severe problem, as complex system designs often consist of a variety of different diagram types interacting with each other. Moreover, often these approaches address specific verification tasks only.

In order to overcome this, the idea of a generic verification framework has been presented in [35]. Instead of considering each description mean separately, the underlying idea of this framework is a transformation into a uniform/normalized description: a *base model*. Moreover, in the course of this transformation complex language constructs are expressed by means of a small subset of so-called *core elements* in order to interface with a wide range of verification engines with complementary strengths and weaknesses. The base model is integrated in the validation and verification process in a way that the designer does not need to have knowledge of it. The results of verification engines which are derived using the base model are mapped back to the source model and represented to the developers in their domain.

In this work, we provide a detailed introduction of the proposed base model and its core elements. Roughly speaking, the base model is a UML class diagram enriched with OCL constraints with a reduced feature set that only contains essential and atomic language constructs. However, we will show that this reduced feature set is sufficient to express many complex language constructs by providing the corresponding model transformations. We focus on transformations within class diagrams because transformations from alternative diagram types, e.g. sequence diagrams or activity diagrams, to class diagrams have already been considered, e.g. in [20] and [19], respectively.

The remainder of this work is structured as follows: motivation for the generic verification framework and a discussion about related work is presented in Sect. 2. A detailed introduction of the base model and its core elements is provided in Sect. 3, while the actual transformations of complex class diagram features into the base model are discussed in Sect. 4. Finally, we conclude the paper in Sect. 5.

2 Motivation and Related Work

The development of automated methods for the verification of UML/OCL models has intensely been considered by researchers and engineers in the recent past. For this purpose, several solving techniques have been applied ranging from a guided enumeration, as done e.g. in the *UML-based Specification Environment* (USE, [14]) together with the language ASSL [13], to the application of verification engines such as CSP solvers [5], Alloy [1], or SAT solvers [34,33]. Fig. 1 gives an (incomplete) overview of the current state-of-the-art categorized by their respective support for diagram types and verification task. While this led to a variety of powerful tools and methods for the verification of UML/OCL models, the resulting state-of-the-art suffers from three main drawbacks:

1. The resulting solutions often support dedicated verification tasks only. While e.g. [34] allows for consistency checking of class diagrams, this approach does

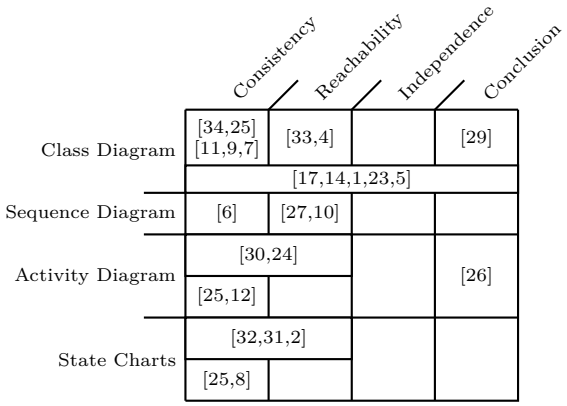


Fig. 1. Overview on Related Work

not support sequence diagrams. That is, for each modeling method and each verification task usually a different verification approach has to be applied.

2. Complex systems are usually not modeled by means of single diagrams only, but composed of a variety of different diagram types which interact with each other. For example, while class diagrams specify the structure of a system, the behavior may be defined by a statechart. But again, most of the available verification approaches support single diagram types only.
3. Almost all proposed verification approaches are bound to one particular verification engine. For example, the approach presented in [5] exploits CSP solvers, whereas e.g. in [33] SAT and SMT solvers find application. This is disadvantageous as verification engines may behave differently effective for various models. If additionally, new and better verification engines emerge in the future, existing transformations to the respective solver input have to be re-developed.

In order to overcome these drawbacks, a generic verification framework for UML/OCL models was envisioned in [35]. The general idea is sketched in Fig. 2: Instead of treating single diagram types separately (as it has mainly been done in the past; see Fig. 1), it was proposed to transform them to a so-called *base model* – a subset of UML/OCL constraints which is expressive enough to describe most constructs of the UML and OCL, but small enough to allow for a flexible further processing.

Having this generic description, the development of verification approaches can focus on the core constructs available in the base model. This allows for an easier integration of verification engines than before. Moreover, even new solving technologies which may emerge in the future can be exploited more easily since a restricted subset of constructs needs to be considered only. In contrast, transformations from the original description means (class diagrams, sequence diagrams, etc.) to the base model have to be provided. But since those would only require a model-to-model transformation to the base model (and e.g. not to the numerous solver-specific inputs), they need to be developed only once.

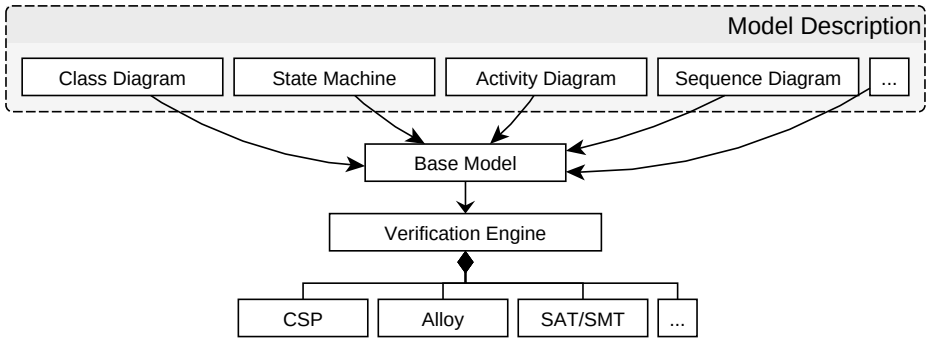


Fig. 2. General Idea of a Generic Verification Framework

By this separation of concerns, a more generic verification of UML/OCL models relying on a variety of solving techniques as well as supporting a wide range of verification tasks, becomes possible at moderate costs.

However, while the principle feasibility of the vision has been demonstrated on selected examples in [35], no implementation of the generic framework exists yet. In particular, a precise definition of the base model and a corresponding transformation scheme from arbitrary description means are still missing[1]. In the following, we aim for closing this gap. More precisely, we provide a precise proposal for a base model and discuss how general constructs can be transformed into it.

3 A Base Model for UML and OCL Verification

As motivated in the previous section, the purpose of the base model is to provide a generic interface between heterogeneous UML/OCL model descriptions and validation and verification tools. This interface shall be flexible and generic at both the source and the target side. More precisely, it shall be applicable for a large variety of diagram types and verification tasks on the one hand, while, on the other hand, it shall allow for a flexible choice of verification engines for further processing. To this end, the base model needs to be:

- *universal*, i.e. for each construct in a source model, an equivalent formulation in the base model must exist, and
- *atomic*, i.e. the constructs of the base model should be limited to fundamental modeling concepts such that a uniform further processing as well as the flexibility of the overall framework is ensured.

Clearly, these are contrary properties as UML and OCL are very powerful languages with a rich set of language constructs – some of which are very complex in nature and can hardly be expressed by simpler means. Consequently, the solution is necessarily a trade-off between universality and atomicity. Nonetheless, we

[1] First ideas, leaving numerous details open, have been sketched in a preliminary version of this paper which has been discussed at a workshop [21].

Table 1. UML Elements in the Base Model

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

✓ core element; ○ transformed element (using only core elements); × unsupported element

aim to reduce the restrictions to universality to a minimum by (a) employing the power of model-to-model transformations on the UML/OCL level and (b) only excluding less relevant UML/OCL features that are hardly used in practice or conceptually infeasible to be tackled by verification engines at all. Note, however, that some restrictions are inevitable and have to be applied anyhow when considering validation and verification of UML/OCL models as, e.g. data types like Integer are unbounded in the standard UML semantics, while verification engines often only work on bounded, finite search spaces. These simplifications are mainly justified by the fact that actual implementations of the models will also have to run on finite resources.

For the foundation of the base model, we propose to use a reduced UML class diagram. This diagram type is well-suited as it natively supports structural definitions in form of classes and associations as well as model dynamics using OCL expressions for operation pre- and postconditions. Furthermore, transformations from other diagram types (such as sequence diagrams or activity diagrams) to class diagrams have already been investigated [20,19] and can be re-used here.

The feature sets of UML and OCL are reduced to a required minimum. This reduction has a few advantages to it: Flexibility and compatibility to verification engines is increased, because the feature set which needs to be supported by it is minimized. In addition, the reduction also enforces an early/high-level transformation of complex constructs into simpler ones which simplifies the analysis of the model and the determination of an appropriate verification engine. In the following, the elements of the reduced feature set of the base model are presented and described in more detail.

3.1 UML Elements in the Base Model

An overview on the different UML class diagram features and how they are included in the reduced feature set of the base model is given in Table 1. The core of the base model is formed of essential and atomic constructs – the so-called *core elements* which are marked with a "✓" and are natively supported in the base model. These have been chosen due to their fundamental importance for UML class diagrams and good compatibility with state-of-the-art verification approaches. Note that for a verification engine to support the base model,

corresponding translations to the solver level need to be developed for these core elements only.

Further class diagram features that can be expressed within the base model, but do not appear in it as core elements, are marked with a "∘". These can be transformed into semantically equivalent representations using only core elements. Details about these transformations will be presented in Sect. 4.

The last category of elements are marked with "×" and are neither part of the base model nor do we propose a corresponding transformation for them yet. These are either (1) hardly used in practice (like Redefines, Subsets, and Union), or (2) are conceptually infeasible for verification engines anyway (like recursive and nested operations).[2] Consequently, the exclusion of these elements only has a minor impact on the universality of the base model.

3.2 OCL Elements in the Base Model

As for OCL, it is a lot harder to reduce the feature set without losing universality. This mostly results from the fact that OCL is a rich language with many diverse operations. Most operations can only be expressed by similar operations or their negated counterpart, e.g. the collection operations `C→isEmpty()` and `C→notEmpty()` can be represented using the operation `C→size()`, and `C→reject(expr)` can be represented using `C→select(not expr)`. A promising candidate to replace many of the standard OCL collection operations, the `iterate` operation, is, however, one of the least supported operations by verification engines – due to its high versatility. Thus, it also does not provide a satisfying solution regarding the reduction of OCL features in the base model.

Our solution is to accept the majority of standard OCL operations in the base model, keeping known alternatives at hand. Then, a verification engine is chosen based on the needs of the model, i.e. one that supports all employed operations or transformed alternatives, and the base model is prepared to be compatible before given to the verification engine.

Besides OCL operations, also data types have to be considered. We propose to use Integers and Sets as core data types. Integers are the mostly used primitive data type and often sufficient to emulate the functionality of other primitive data types like enumerations, Reals, and other numeric data types. Even Strings (on the word-level rather than on the character-level) may be emulated by Integers. Likewise, Sets are a well-suited representative for collection types. Besides that, other collection types like Sequence can also be emulated using UML classes as will be outlined later in Sect. 4 – although with considerable overhead.

Overall, data types are interpolated if necessary and a large set of OCL operations is accepted in the base model – even if the particular set of operations may possibly restrict the set of appropriate verification engines to be used for further processing. Verification engines used in combination with the base model are expected to support at least basic arithmetic on `Set` and `Integer` as well as the quantifiers `forAll` and `exist` (preferably also the `closure` operation).

[2] Note however that OCL provides the closure operation (which can solve some tasks that are typically formulated recursively) and which is supported by our approach.

4 Transformation to the Base Model

This section defines the transformation of class diagram features in UML/OCL source models into the target base model. The transformation consists of many smaller UML and OCL model transformations, some of which have already been sketched [18]. We focus on selected transformations (○ elements) from Table 1 that show the concepts of the base model best.

All transformations shown in this section operate on the UML and OCL layer. Where appropriate, we use instances of the UML metamodel to illustrate the transformation, showing system states before and after the transformation. Figure 3 shows the relevant parts of the UML metamodel used in the transformations. In the top left corner, the class `Element` is located, defining the base of every element in the model and on the right side you see the generalization hierarchy originating from it, defining different abstractions used by several elements throughout the metamodel. Finally, in the lower left part of the figure, the classes `Class`, `Association` and `AssociationClass` are defined extending the general class `Classifier`. These elements are connected via the class `Property` to define, e.g. roles and attributes. Note that transformations mostly concerning OCL expressions are not shown in the UML metamodel.

The transformation definitions on the UML metamodel are required to implement the transformations using tools like QVT [28] or ATL [22]. Along with the transformations, some further aspects have to be considered for the base model to work properly:

Tracability As mentioned earlier, the base model is a "bridge" between the developer's source model and the verification engine, and results from the

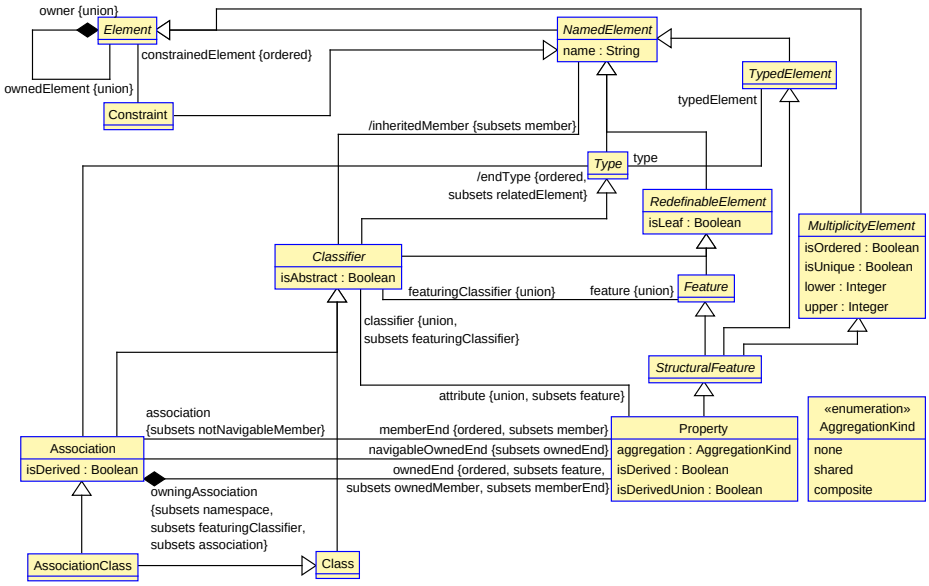


Fig. 3. Excerpt of relevant parts from the UML Metamodel

verification engine are presented in terms of the source model. Therefore, tracability is an important requirement for the base model. It has to be possible to transform models into the base model and solutions found by verification engines back into the source model. The easiest solution is to use bidirectional transformation methods (e.g. QVT relational [28]) and delegate the tracing to their built-in methods. However, the difference in the meta layers of the transformations usually require further adjustments. Additionally, some transformations (mainly those operating on the UML layer) are simple enough to be traced by the names of the elements involved, e.g. an invariant name hinting at the corresponding element in the source model. Finally, to be consistent with the base model idea, having all information in the UML/OCL model, UML comments can be applied during the transformation to provide further tracing information.

Equivalence While general interactive model verification techniques are in principle available [3], we propose to check transformation equivalence by automatically building test cases. As many of the transformations work on the UML metamodel, transformation test cases in form of object diagrams can be constructed by instantiating the left and right hand side of the transformation. Afterwards the desired equivalence properties are checked by formulating OCL properties on the union of left and right hand side as has been studied in [15,16]. While we know the importance of these tests, in this paper, we do not study them in detail and focus on the transformations.

4.1 Transformation of Ternary (n-Ary) Associations

A rather simple model transformation is the replacement of ternary associations by a class and binary associations plus constraints. Figure 4 gives an overview of the transformation. The source model is on the left having a ternary association `ABC` connecting three classes. The model after the transformation is shown on the right side. Instead of the association, there is a new class named `ABC` connecting the three classes using three binary associations. The role names are carried over for the corresponding association ends and new ones are added where necessary. By this, ternary associations can be transformed into core elements.

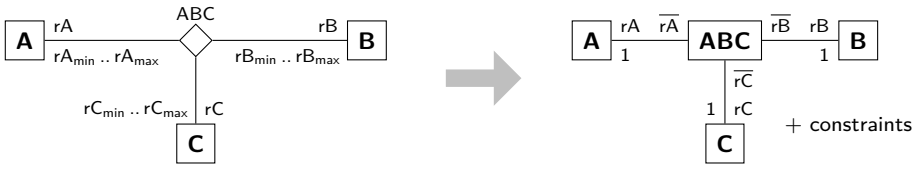


Fig. 4. Class Diagram View of Ternary to Binary Association Transformation

Figure 5 shows the same models as instances of the UML metamodel[3]. The ternary association form the left-hand side is located in the top left area of

[3] Many attributes, (derived) links and objects are not relevant for the transformation and, hence, are hidden for a better overview and understandability.

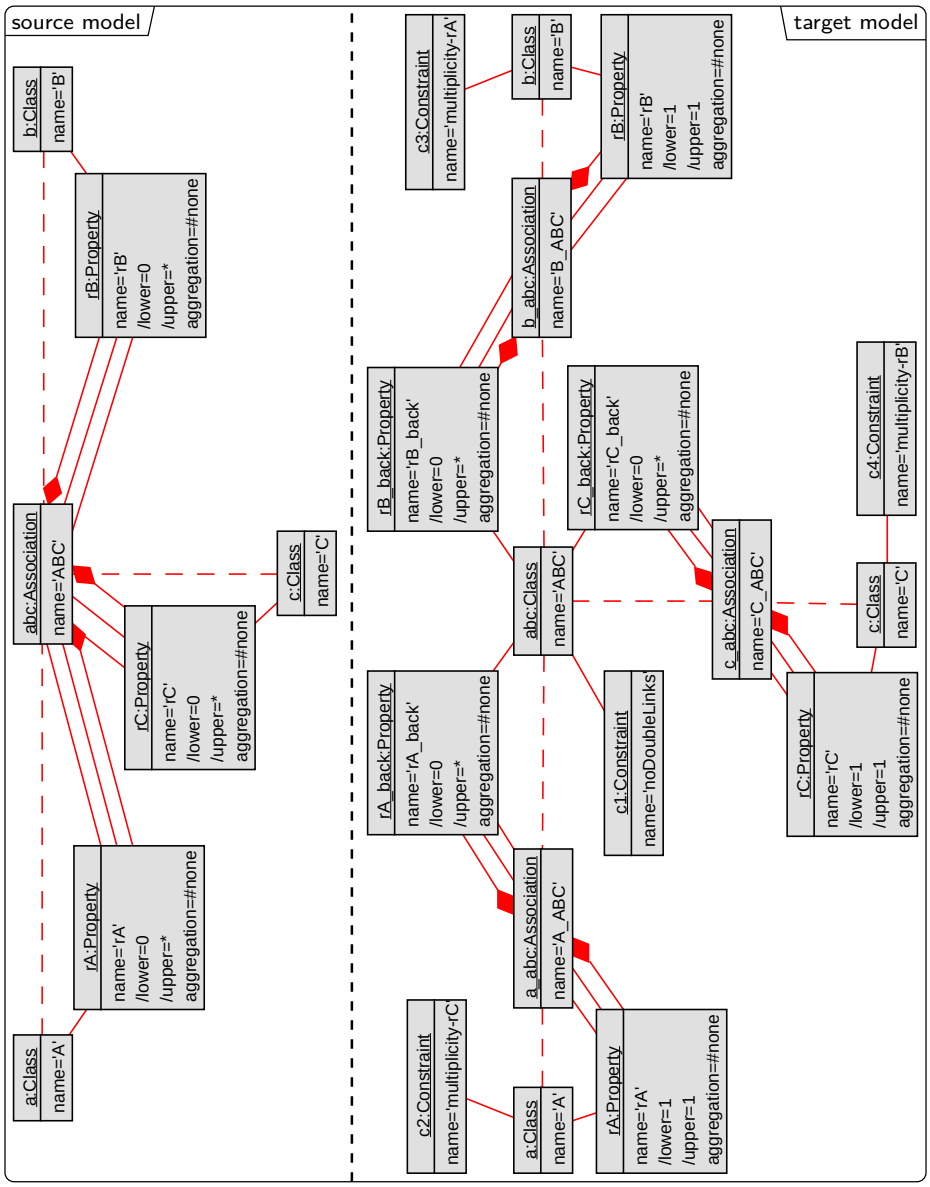


Fig. 5. Ternary Association to Class plus Binary Associations in UML Metamodel

the picture with the association object *abc* and its three connected classes. The role information is contained in the `Property` objects connected in between the classes and the association. The multiplicities shown by the attributes `/lower` and `/upper` for the roles are derived values from elements hidden in the figure. The links between the association and the properties define ownership and navigability as present in the metamodel. Dashed lines symbolize derived

links, showing relations between objects that are indirectly related via other objects, i.e. `Property` objects. In the UML metamodel from Fig. 3, these links are instances of the association going from the class `Association` upwards and right to the class `Type` (role `/endType`), showing the `Type` objects linked with the association. For example, the derived link between the objects *a* and *abc* offers direct access to one of the end types of association *abc*.

In the lower right of Fig. 5, the right-hand side of the transformation is shown. The association object *abc* was transformed into a class and three new associations are created. The original properties are still present, however the multiplicities are changed and properties have been added to fill the missing roles. Furthermore, to ensure semantic equivalence between the models from the left and right hand side, two types of constraints, representing the properties of the ternary association, are added to the classes. First, three objects (*a*, *b*, *c*) can only be connected once by the association `ABC`. And second, multiplicities for the roles of the ternary association have to hold, i.e. the number of pairs of objects *b* and *c* that are connected to *a* objects must be within the specified multiplicity of role `rA`. If a multiplicity is specified as `0..*`, no constraint is required. The following two invariants exemplify these two properties:

```
context r, r' : ABC inv noDoubleLinks:      -- one link per tuple (A,B,C)
( r.rA = r'.rA and r.rB = r'.rB and r.rC = r'.rC ) implies r = r'

context b : B inv multiplicity-rA:          -- multiplicity for role rA
C.allInstances()→forAll( c | let linkCount =
    ABC.allInstances()→select( r | r.rB = b and r.rC = c )→size()
  in linkCount >= rA_min and linkCount <= rA_max )
```

Finally, all expressions refering to role navigations that are now transformed have to be adjusted. Also, while this example concentrated on a ternary association, the concepts are applicable to *n*-ary associations with more than three association ends as well.

4.2 Transformation of Association Classes

Next the transformation of UML association classes into base model compatible description means is considered. Existing model transformations suggest the conversion into ternary associations plus OCL constraints, however only binary associations are allowed in the base model. To overcome this issue, the transformation rules for the base model can be combined together to sequentially transform the source model into a proper base model, i.e. after the transformation into the ternary association the transformation from the previous section is able to simplify it into binary associations.

The transformation from association classes into ternary associations is depicted in Fig. 6. The class and association information is split into a class and an association. Implicit definitions are made explicit, e.g. the implicit role name `C` in the source model for the association class has been made explicit on the right side. The semantic properties are expressed with OCL constraints. Figure 7 shows the transformation with instances of the UML metamodel. The separation of the association class into class and association is clearly visible.

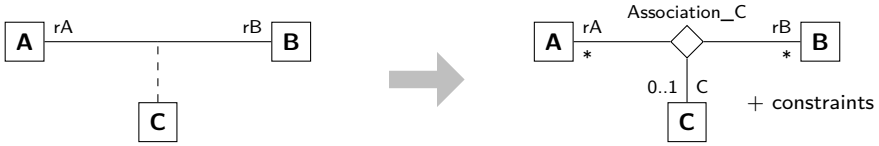


Fig. 6. Association Class to Ternary Association Transformation

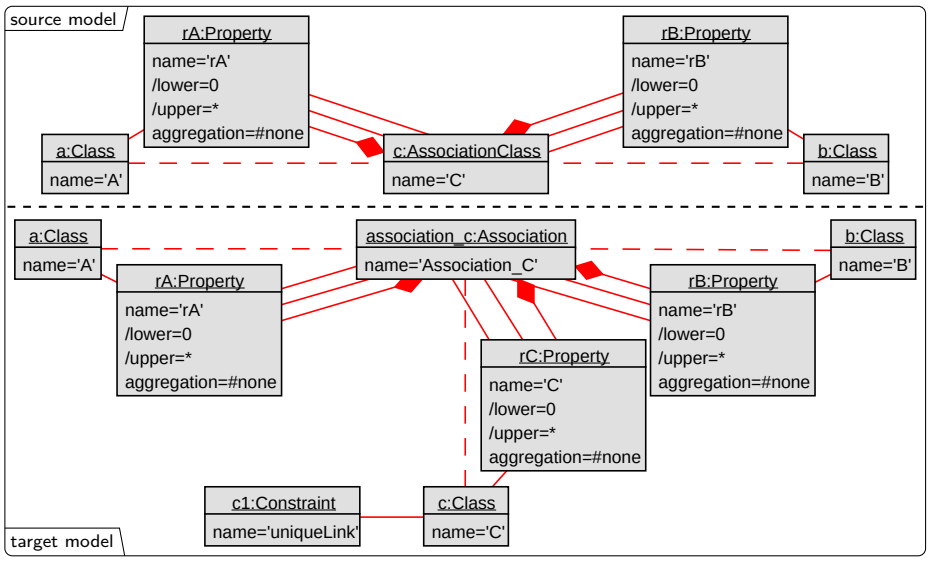


Fig. 7. Association Class to Class plus Ternary Association in UML Metamodel

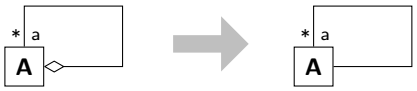
Similar to the previous section, the association class has properties that this transformation has to adhere to. In particular, for every two objects a and b there may be at most one association class linking them. The following invariant is showing this property as an OCL invariant:

```
context c : C inv uniqueLink:                    -- one link per pair (A,B)
c.rA→size() = 1 and c.rB→size() = 1 and C.allInstances()→forAll( c' |
    (c.rA = c'.rA and c.rB = c'.rB) implies c = c' )
```

Finally, multiplicity constraints and adjustments to other OCL expressions relying on transformed navigations are similar to those of the transformation in the previous section.

4.3 Transformation of Aggregations and Compositions

Aggregations and compositions are special types of associations that specify a whole-part relationship. They have unique properties that distinctly define their semantics within the class diagram. However, these properties are not explicitly modeled in the UML metamodel, only the enumeration attribute `aggregation` of the class `Property` indicates whether an association is treated as an aggregation or composition. Thus, the transformation in the UML structure is trivial.




```
context A inv:
self.a→closure( a )→excludes( self )
```

Fig. 8. Aggregation Example and its Semantics Expressed in OCL

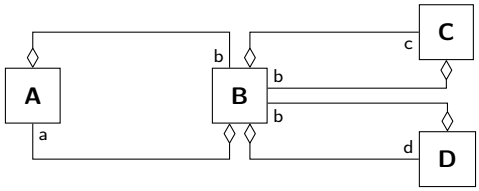


Fig. 9. Complex Aggregation Model

The challenge is to express the inherent properties as OCL constraints. In the following, we will illustrate the generic transformation by means of examples.

The property of aggregations define that an aggregate cannot be part of itself (*cycle freeness*), i.e. navigating to a part results in "smaller" objects than the whole. Looking at compositions, a few more properties exist: Each part may at most have one whole (*forbidding sharing*); and when a composite is destroyed all its connected parts are destroyed as well, thus the composite is responsible for its parts (*ownership*).

The cycle freeness property is a special one because it can be affected by multiple aggregations at once. In the easiest case, a reflexive aggregation forms a cycle as pictured in Fig. 8 and the corresponding OCL expression to describe the property is straightforward. However, cycles can span over an arbitrary amount of links and the complexity of the OCL expression rises with the amount of aggregations connected. These constellations are commonly found in metamodels, when an aggregations connect elements related via a generalization hierarchy.

As an example, consider the model in Fig. 9. At a first glance there are three independent cycles. But these cycles all share a common class `B`. This affects how cycles can occur in a system state. For example, starting from class `A`, a cycle can be as simple as connecting to a `B` object and back to the original `A` object. However, since `B` is also connected to `C` and `D`, there can be an arbitrary amount of links in between. A single `closure` operation, as pictured in Fig. 8, does not cover all possible cases allowed by the class diagram. To consider all paths going from class `B` to itself, a second, *nested* `closure` operation is required. The full invariant ensuring cycle freeness for class `A` looks as follows:

```
1 context self : A inv:
2 self.b→closure( c.b→union( d.b ) ).a
3   →closure( b→closure( c.b→union( d.b ) ).a )→excludes( self )
```

The repeating[4], highlighted expression in lines 2 and 3 is the essential part. Instead of only considering all navigations from class `A` to `B` and back, the nested

[4] The lack of a non-reflexive transitive closure operation in OCL forces the repetition of the expression here.

`closure` operations (line 3) cover all intermediate navigations from class `B` to itself. Note that not all cycles of class `B` are inside the nested `closure` expression, since the one between classes `A` and `B` is already covered by the initial navigation.

The previous examples for cycle freeness shown with aggregations are also valid for compositions. Additionally, the other previously defined properties have to be considered. Figure 10 shows an example for the *forbidding sharing* property as an OCL invariant. The constraint ensures that none or exactly one of the possible composites is linked with every `Part`, thus preventing multiple links at the same time.

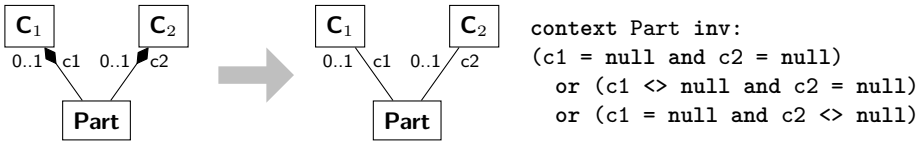


Fig. 10. Composition Forbidding Sharing Property Expressed in OCL

Finally, the *ownership* property is left. This property is different from the previous ones, since it defines behavior, e.g. during operation calls, instead of structure. Therefore, this property cannot be expressed as a structural invariant. To represent it in a class diagramm, we use operation pre- and postconditions. However, since the full transformation requires different description means not discussed in this work, we leave the details for future work.

4.4 Transformation of Query Operations

Query operations are side-effect free OCL expressions assigned to classes as operations. The transformation of such operations in the base model is mainly operating on OCL expressions. In general, all calls to the query operation in any expression can be expanded into the expression associated with the operation. Parameter expressions are obtained from the respective operation call and the return value is simply the result of the expression.

In case of recursively defined query operations, the expansion never terminates. However, a general idea for these situations is to transform the expressions into closure expressions or expand the expression a fixed amount of times, depending on the estimated requirements, but this approximation is not always possible. Nevertheless, in terms of compatibility and performance, this transformation has next to no drawbacks.

4.5 Transformation of OCL Collection Types

In case that a verification engine cannot be used for a certain source model – due to incompatibilities on the OCL level, e.g. an unsupported collection type – a last resort can be the representation of such a type in the class diagram itself. Figure 11 shows a class with a `Sequence` typed attribute on the left side. The resulting model is extended by a (simplified) representation of such sequence as classes in the diagram. The sequence is connected to the class `IntegerValue`,

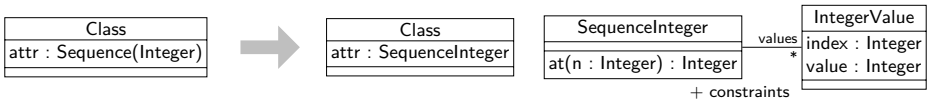


Fig. 11. Sequence Transformation into UML Structure

which has an index and the actual integer value. Constraints are applied to ensure semantics, e.g. a well-defined order exists.

The query operation `SequenceInteger::at(Integer)` shows an example for the transformation of the functionality of the modelled type. The OCL definition looks as follows:

```
context SequenceInteger::at( n : Integer ) : Integer =
  self.values→any( index = n ).value
```

Other common standard OCL operations can be defined accordingly. Also, the definition of a sequence is reusable for multiple occurrences of the same type in the source model.

This transformation is mostly interesting when no verification engine is able to handle a given model without this transformation. While the overhead is considerable, being able to apply validation and verification techinques to a previously incompatible model demonstrates the universality of the base model.

4.6 Combination of the Transformation Concepts

Using the transformations discussed above, source models can be transformed into base models by applying the transformations until no further matches can be found. That way, the resulting model consists of *core elements* (✓) only, while the semantics of all *transformed elements* (○) is preserved (see Table 1). Along with the transformation of the respective model elements, all OCL expressions using these elements are transformed as well, to match the modifications.

5 Conclusion and Future Work

In this paper, we have proposed a transformation of UML/OCL models to base models. By this, we closed a significant gap for generic UML and OCL model validation and verification. The base model increases compatibility between source models and verification engines, by unifying various diagram types and expressing them using a reduced feature set, the so-called core elements. The result is a universal base model consisting of atomic elements only. We have also presented the corresponding transformation concepts for an important set of complex UML/OCL constructs like association classes, compositions, and OCL collection types. In order to transform a given source model into the corresponding base model representation, transformations are applied successively until the model only consists of core elements.

When in the future, verification engines support more complex features directly, it might be preferable to use those direct translations instead of performing the proposed transformations. However, an evaluation of the performance gain of direct translations by the verification engines versus the base model

transformations is left for future work. If case studies reveal benefits for chosing different core elements, the base model can easily be adapted, due to the modular combination of transformations. Finally, transformations are required to be able to map verification results on the base model back to the source model.

Acknowledgement. Thanks to Lars Hamann for the constructive discussions about the model transformations, in particular the aggregation transformation. We also thank the reviewers for their useful feedback. 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.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: MoDELS. pp. 436–450. Springer (2007)
2. Banerjee, A., Ray, S., Dasgupta, P., Chakrabarti, P.P., Ramesh, S., Ganesan, P.V.V.: A Dynamic Assertion-Based Verification Platform for Validation of UML Designs. ACM SIGSOFT Software Engineering Notes 37(1), 1–14 (2012)
3. Brucker, A.D., Wolff, B.: Semantics, calculi, and analysis for object-oriented specifications. Acta Inf. 46(4), 255–284 (2009)
4. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL Operation Contracts. In: IFM. pp. 40–55. Springer (2009)
5. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software 93, 1–23 (2014)
6. Chen, Z., Zhenhua, D.: Specification and Verification of UML2.0 Sequence Diagrams Using Event Deterministic Finite Automata. In: SSIRI. IEEE (2011)
7. Chiorean, D., Pasca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML Models Consistency Using the OCL Environment. Electr. Notes Theor. Comput. Sci. 102, 99–110 (2004)
8. 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)
9. Demuth, B., Wilke, C.: Model and Object Verification by Using Dresden OCL. In: IIT-TP. p. 81. Technical University (2009)
10. Dinh-Trong, T.T., Ghosh, S., France, R.B., Hamilton, M., Wilkins, B.: UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs. In: Storey, M.D., Burke, M.G., Cheng, L., van der Hoek, A. (eds.) ETX. pp. 120–124. ACM (2005)
11. Duran, F., Gogolla, M., Roldan, M.: Tracing Properties of UML and OCL Models with Maude. In: AMMSE. pp. 81–97. Electr. Proc. Theor. Comput. Sci. (2011)
12. Eshuis, R., Wieringa, R.: Tool Support for Verifying UML Activity Diagrams. IEEE Trans. Software Eng. 30(7), 437–447 (2004)
13. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling 4(4), 386–398 (2005)
14. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based Specification Environment for Validating UML and OCL. Sci. Comp. Prog. 69(1–3), 27–34 (2007)
15. Gogolla, M., Hamann, L., Hilken, F.: Checking Transformation Model Properties with a UML and OCL Model Validator. In: Amrani, M., Syriani, E., Wimmer, M. (eds.) VOLT@STAF. pp. 16–25. CEUR Proc., Vol. 1325 (2014)
16. Gogolla, M., Hamann, L., Hilken, F.: On Static and Dynamic Analysis of UML and OCL Transformation Models. In: Dingel, J., de Lara, J., Lucio, L., Vangheluwe, H. (eds.) Analysis of Model Transformations (AMT). CEUR Proc., Vol. 1277 (2014)

17. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) Tests and Proofs (TAP). pp. 90–104. Springer, Berlin, LNCS 5668 (2009)
18. 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)
19. Hilken, C., Seiter, J., Wille, R., Kühne, U., Drechsler, R.: Verifying Consistency between Activity Diagrams and Their Corresponding OCL Contracts. In: Forum on specification & Design Languages (FDL) (2014)
20. Hilken, C., Peleska, J., Wille, R.: A Unified Formulation of Behavioral Semantics for SysML Models. In: Modelsward (2015)
21. Hilken, F., Niemann, P., Wille, R., Gogolla, M.: Towards a Base Model for UML and OCL Verification. In: Boulanger, F., Famelis, M., Ratiu, D. (eds.) MoDeVVa@MODELS. pp. 59–68 (2014)
22. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
23. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: TOOLS. pp. 290–306. Springer (2011)
24. Kurth, F., Schupp, S., Weißleder, S.: Generating Test Data from a UML Activity Using the AMPL Interface for Constraint Solvers. In: Tests and Proofs (TAP). LNCS, vol. 8570, pp. 169–186 (2014)
25. Kuske, S., Gogolla, M., Kreowski, H.J., Ziemann, P.: Towards an Integrated Graph-Based Semantics for UML. Softw. and Sys. Modeling 8(3), 403–422 (2009)
26. Lam, V.S.W.: A Formalism for Reasoning about UML Activity Diagrams. Nordic Jrnl. of Comp. 14(1), 43–64 (2007)
27. Lima, V., Talhi, C., Mouheb, D., Debbabi, M., Wang, L., Pourzandi, M.: Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages. Electr. Notes Theor. Comput. Sci. 254, 143–160 (2009)
28. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.1 (january 2011) edn., `http://www.omg.org/spec/QVT/1.1/`
29. Queralt, A., Teniente, E.: Reasoning on UML Class Diagrams with OCL Constraints. In: Conceptual Modeling – ER. pp. 497–512. Springer (2006)
30. Rafe, V., Rafeh, R., Azizi, S., Miralvand, M.R.Z.: Verification and Validation of Activity Diagrams Using Graph Transformation. In: ICCTD. pp. 201–205. IEEE (2009)
31. Rodríguez, R.J., Fredlund, L., Herranz-Nieva, Á., Mariño, J.: Execution and verification of UML state machines with erlang. In: Giannakopoulou, D., Salaün, G. (eds.) Software Engineering and Formal Methods. pp. 284–289 (2014)
32. Schwarzl, C., Peischl, B.: Static- and Dynamic Consistency Analysis of UML State Chart Models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MoDELS. pp. 151–165. Springer (2010)
33. Soeken, M., Wille, R., Drechsler, R.: Verifying Dynamic Aspects of UML Models. In: DATE. pp. 1077–1082. IEEE (2011)
34. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL Models Using Boolean Satisfiability. In: DATE. pp. 1341–1344. IEEE (2010)
35. Wille, R., Gogolla, M., Soeken, M., Kuhlmann, M., Drechsler, R.: Towards a Generic Verification Methodology for System Models. In: DATE. pp. 1193–1196 (2013)