Using MDE for Generic Comparison of Views

Bas Graaf[1] and Arie van Deursen[1,2]

[1] Delft University of Technology, The Netherlands
b.s.graaf,arie.vandeursen@tudelft.nl
[2] CWI, The Netherlands

Abstract. We investigate the application of technologies for model-driven engineering to check the conformance of two software models. This involves their model-based comparison, and visualisation of the results. To generalise our approach we use reflection, metamodel generalisation, and higher-order transformations. We apply our approach to assess the extent to which the implementation of an academic example system does not violate the constraints defined by its architecture specification.

1 Introduction

One aspect of validation and verification is the evaluation of whether an implementation does not violate the constraints defined by its architecture. In this paper we investigate an approach to automate this process, to which we refer as conformance checking.

We assume that for automatic conformance checking the involved artifacts need to be specified in well-defined languages. These artifacts are models of the software under development. In general, these models are on different abstraction levels and might be expressed in different modelling languages. In previous work [1], we proposed an approach for conformance checking that extracts from both the implementation (source code) and an architectural view (UML) a model in the same, intermediate (in terms of abstraction level) language, and subsequently compares the extracted models. We implemented these steps as transformations using XML technology. The use of XML made the specification of these transformations verbose and, hence, difficult to maintain. Moreover, each type of architectural view required a specific implementation.

In this paper, we investigate the use of model-driven engineering (MDE) technologies to check the conformance of software models in a more generic way. The problem of obtaining suitable models from an implementation and an architecture specification for comparison is outside the scope of this paper.

The structure of this paper is as follows. In Sect. 2 we introduce the running example to which we apply our approach. Then, in Sect. 3 we first illustrate how a comparison transformation can be defined for a specific metamodel. Subsequently, we explain how this can be generalised using higher-order transformations (HOTs), reflection, and metamodel generalisation. The result of applying this generic approach are presented in Sect. 4. In Sect. 5 we discuss a number of potential improvements of our approach. We conclude the paper with an overview of related work, our contribution and opportunities for future work.

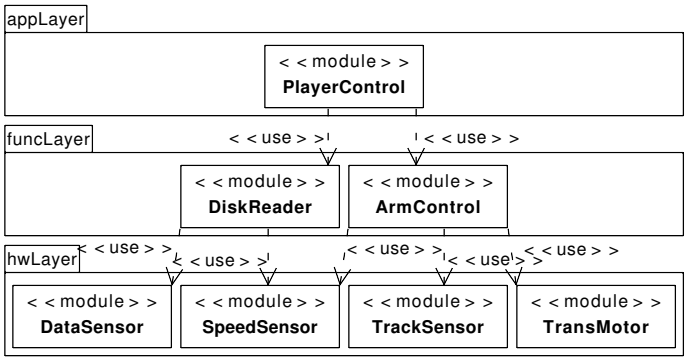


Fig. 1. Module-uses view

2 Running Example

As a motivating application for our model conformance checking approach, we consider the assessment of whether the implementation of a software system conforms to its architecture. A simple system developed in the context of a student project serves as a running example.

The Digital Music Box (DMB) reads data from a paper disc (the record) that contains a plotted spiral track of pulse-width modulated data bits. It rotates with a constant speed. The system tracks the spiral, reads the data bits, and then maps those bits to symbols. A string of symbols will be fed to an output device that transforms the string into audible music. The DMB is composed of a traditional turn table and a set of simple light sensors that can be moved axially by a motor. The control is implemented in Java and distributed between a simple micro controller and a PC.

We created several architectural views (i.e., models) for the development of the DMB, such as module views and component-and-connector views. One of those, the module-uses view, is depicted in Fig. 1. It represents the architecture in terms of implementation units (modules). Modules are represented by UML classes stereotyped ≪module≫, and uses relationships by UML dependencies stereotyped ≪use≫. Layers are represented using UML packages. The lower layers are closer to the hardware (e.g., motors and sensors), while the upper layers are closer to the user.

3 Approach

Our aim is to assess the extent to which DMB's implementation conforms to the architectural model presented in the previous section. To this end, we extract information from the implementation (as discussed in van Dijk et al. [1]) and based on that instantiate a similar model (i.e., in the same language) as depicted in Fig. 1. Next, we compare those models with the architectural models and visualise the result.

Although the architectural model is specified using UML, it only includes a few types of modelling elements (compared to the number of modelling elements

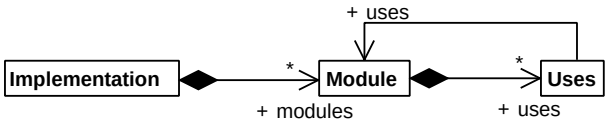


Fig. 2. MADL metamodel

available in UML). This is also true for other types of architectural models (e.g., component-and-connector models). In order to make our comparison transformation not unnecessarily complex, we decided to define a metamodel for each type of model. We define these metamodels such that it is possible to instantiate a conforming model from both the corresponding architectural (UML) models as well as the implementation. Actually, such metamodels define an architecture description language (ADL). For example, the metamodel of the language for model-uses view, Module ADL (MADL), is depicted in Fig. 2. It is defined using the MetaObject Facility[1] (MOF), a language for defining metamodels.

The MADL metamodel defines one root element, which is named Implementation because a module view depicts a system as a set of implementation units. An Implementation consists of Modules that own any number of Uses elements representing use relationships. Note that we defined the Uses as a MOF Class instead of a Reference, which we explain later. Our comparison transformation for module-uses models takes models conforming to this metamodel as input.

For the specification and implementation of the comparison model transformation, we use the Atlas Transformation Language [2] (ATL). This model transformation language uses the Object Constraint Language[2] (OCL) to match source model elements and to specify what target model elements are to be created for each matched source model element, and how its features are to be instantiated. Additionally, it is possible to define helper attributes and operations that extend the set of features of an element as defined by its metamodel.

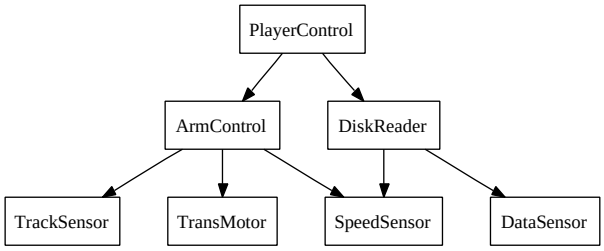
3.1 Naive Approach

We assume that we are able to derive MADL models from both the architectural views and the implementation. They are visualised in Fig. 3. A Module is represented by a box, and a Uses dependency by an arrow. Note that Fig. 3(a) is a simplified representation of Fig. 1.

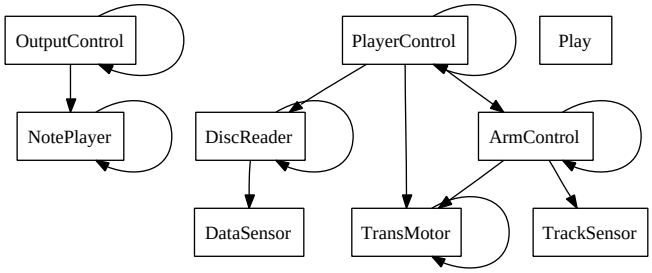
We define a transformation that takes the two models that need to be compared as source models (i.e., an architecture and an implementation model). The target model contains the union of the elements and relations in the source models, labelled according to their occurrence in either one or both of them. We use the labels introduced by Murphy et al. [3]:

convergent element or relation present in both models;
absent element or relation present only in the architecture model; and
divergent element or relation present only in the implementation model.

[1] `http://www.omg.org/mof`
[2] `http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL`



(a) Architecture



(b) Implementation

Fig. 3. Reconstructed MADL models

To allow this labelling we define the metamodel element ViewElement that has a conformance attribute of the enumeration type Conformance. We defined the latter to encompass the literals convergent, absent, and divergent (see also Fig. 4). The elements of the metamodel in Fig. 2 are defined as specialisations of this ViewElement element. By having a common generalisation for all such elements we can employ reflection to generalise some of the helpers and rules we need to define. This was also the reason to define the Uses relation as a MOF Class instead of a Reference; the latter cannot be generalised or have attributes (i.e., it is not a first-class element).

To establish the status of a ViewElement in the implementation or architecture models, we use OCL expressions as in Listing 1.1. This helper can be understood as follows. The `allInstancesFrom()` helper selects all instances of the specified metamodel element from the named metamodel (e.g., the `'ARCH'` metamodel refers to the metamodel associated with the architecture model). The helper defines what a divergent element is, using OCL. It uses two additional helpers. The `isFromImplementation` helper checks whether the element (`self`) was contained in the implementation source model. The `isMapping()` helper operation checks whether the element (`self`) can be mapped to the element passed to it as parameter (`m`). We use reflection (`self.oclType()`) to only select those elements from the architecture that have the same type as the element for which the `isDivergent` helper was called (it makes no sense to compare elements of different types). As a result, we only have to define one (polymorphic) `isDivergent` helper for all ViewElements (in this case: Module and Uses). In summary, this

```
1 helper context MADL!ViewElement def: isDivergent : Boolean =
2  self.isFromImplementation and
3  self.oclType()->allInstancesFrom('ARCH')->forAll(m | not self.isMapping(m));
```

Listing 1.1. isDivergent helper

```
1  rule Uses {
2   from r_in : MADL!Uses (r_in.isConvergent or r_in.isDivergent or r_in.isAbsent)
3   to r_out : MADL!Uses (
4     uses <- r_in.uses.match,
5     conformance <- if r_in.isConvergent then
6      #convergent
7     else if r_in.isDivergent then
8      #divergent
9     else
10     #absent
11    endif endif)}
```

Listing 1.2. Rule for Uses elements

helper defines a divergent element as those elements from the implementation model (line 2), for which all elements in the architecture model are not mappings (line 3).

We defined a similar `isAbsent` helper. For convergence, the corresponding `isConvergent` helper evaluates to true only for convergent elements and relations of *one* of the source models. This is necessary because we use these helpers to specify transformation rules that match convergent, divergent or absent elements and create corresponding elements in the target model. For convergent elements we want such a rule to match only one element of each pair of corresponding convergent implementation and architecture model elements. We arbitrarily chose the implementation model element for this.

Using these helpers we can specify the transformation rules for each of the ViewElements. As an example, Listing 1.2 depicts the rule for Uses elements.

The from part of the rule in Listing 1.2 specifies a filter to ensure that the rule matches all Uses elements that will be labelled convergent (note that this only includes elements from the implementation model), divergent or absent.

The `conformance` feature is initialised according to the status of the matching element (`r_in`) using a conditional expression (lines 5–11).

To understand how the `uses` feature (representing the target of a Uses relation) is set (line 4), we need to explain how ATL initialises bindings of features. To bind a feature to the target model element of another rule, we need to specify the source model element that matches that other rule[1]. The ATL transformation engine then automatically resolves the binding to the corresponding target model element. As such, we specify in the binding of the `uses` feature the element referred to by the `uses` feature of the matching Uses element (`r_in.uses`).

[1] This is caused by the fact that with ATL the target model is read-only, and cannot be navigated

```
1 helper context MADL!Module def: isMapping(m : MADL!ViewElement) : Boolean =
2  self.name = m.name;
3
4 helper context MADL!Uses def: isMapping(u : MADL!Uses) : Boolean =
5  self.uses.isMapping(u.uses) and self.user.isMapping(u.user);
```

Listing 1.3. isMapping helpers

Subsequently, we apply the `match` helper to that Uses element (from one of the source models). This is only relevant in the case that the matched Uses element was contained in the architecture model and its `uses` feature refers to a Module that is convergent. The `match` helper then retrieves the corresponding element from the implementation model (otherwise it simply returns `self`).

As a final illustration, Listing 1.3 depicts the `isMapping` helpers for the Module and Uses elements. We say that a Module from the implementation model maps to a Module from the architecture model when both have the same name (line 2). Alternatively, we could rely on a user-provided mapping (e.g., as a separate source model of the transformation).

We use the `isMapping` helper for Module elements to define a corresponding helper for Uses elements. This `isMapping` helper defines that a Uses element maps to another Uses element when the source and target modules of the uses relations they represent map to each other (selected in line 5 by the `user` attribute helper and the `uses` feature, respectively).

3.2 Generalising the Approach

During specification of the conformance transformations for MADL and other view metamodels, we noticed that they consist of similar rules and helpers. They differ with respect to the types and names of metamodel elements, as well as the names and number of their references. We could identify three types of metamodel elements occurring in our view metamodels: root, entity, and relation elements. Examples of these type of elements in our MADL metamodel are Implementation, Module, and Uses, respectively. Each type requires the specification of different transformation rules and helpers depending on the element's name, and the number, names and multiplicity of its references.

The similarity of the rules and helpers for a particular type of metamodel element raises the idea to extend the set of supertypes (beyond ViewElement) to generalise our conformance transformations. One option is to only allow a single generic metamodel that is given different semantics for different views. As this may easily result in misinterpretation, this is not desirable.

Alternatively, we can allow to specialise the generic metamodel for a particular view. Although transformation rules then still need to be specified for every concrete type, some of the helpers can be made generic using polymorphism and reflection. However, this is only possible for helpers that do not navigate a type's references as these can neither be generalised in a MOF metamodel nor discovered using the reflective capabilities of ATL.

As such, by only introducing generic metamodel elements we cannot fully generalise our conformance transformations. This made us investigate whether it is possible to generate such a transformations automatically from a metamodel. In principle ATL is particularly suited for this because it has an associated metamodel. Using this metamodel ATL transformations can be represented as models that can be the source and target of transformations themselves. Such transformations are referred to as higher-order transformations (HOTs).

To generate different types of rules and helpers for the different types of elements in a metamodel (root, entity, and relation), we specialised the generic ViewElement we defined earlier, resulting in the (abstract) metamodel elements depicted in Fig. 4. The root element of a metamodel for a view will extend ViewRoot; first-class elements will extend ViewEntity; and relations will extend ViewRelation.

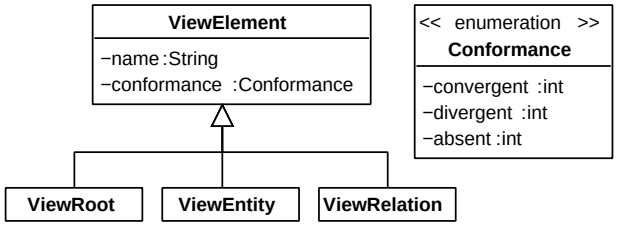


Fig. 4. Generic metamodel elements

In the case of MADL, Implementation is a ViewRoot, Module is a ViewEntity, and Uses is a ViewRelation. Using these generalisations and reflection we are able to implement such a HOT[1]. It can be used in the following procedure.

First define a metamodel for a view using KM3 [4], a simple, textual language for the definition of metamodels. The elements of this metamodel should specialise those depicted in Fig. 4. The rest of the procedure is completely automated using Ant[2]. We inject the KM3 file (textual) into a KM3 model. This model serves as source for the transformation we defined. The target of this transformation is a model of an ATL transformation. We extract this target model into an ATL file (textual), which defines a transformation that takes two models conforming to the metamodel as source models and generates a target model in which all model elements are labelled according to their conformance. Our procedure reuses the metamodels for KM3 and ATL, as well as corresponding injectors and extractors from the ATL Development Toolkit.

4 Results

We tested our HOT on the MADL metamodel by applying the resulting ATL transformation to the models that were depicted in Fig. 3. The result is visualised

[1] This transformation and the example from this paper are available at `http://www.eclipse.org/m2m/atl/atlTransformations/#KM32CONFATL`

[2] A Java-based build tool (`http://ant.apache.org/`)

in Fig. 5. It shows that the model elements are correctly labelled. Convergent, absent, and divergent elements and relations are represented using solid, dotted, and dashed lines, respectively. Note that some inconsistencies do not indicate serious problems, for instance, those resulting from simple name mismatches.

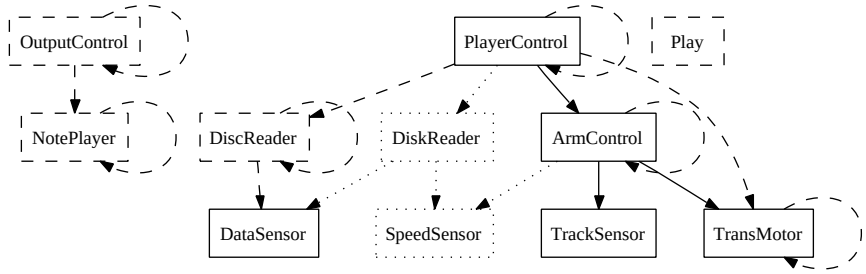


Fig. 5. Merged MADL conformance model

5 Discussion

Although in our example the MADL metamodel only contains a single concrete metamodel element for each of the generic types, our HOT also allows metamodels for which this is not the case. For a metamodel that defines multiple types of modules or uses (or other type of) relations, it generates separate transformation rules and helpers.

The generic metamodel elements we defined are aimed at one category of architectural views, that is, module views. However, Clements et al. [5] define a second major category of architectural views: component-and-connector views. These views have more complicated metamodels that include elements such as, system, component, connector, port, and role. Without going into details, in such views a system consists of components that have ports, and connectors that have roles. Two components can interact by *attaching* their ports to compatible roles of a connector. Some of these elements specialise the generic elements we already defined (e.g., a component is a ViewEntity, a system is a ViewRoot, and an attachment is a ViewRelation). For others we need to extend our approach. For a connector, for instance, which is a first-class entity in such views and therefore is not a ViewRelation, we could define a new generic type: ViewConnectingEntity. For such elements different types of rules and helpers can then be created. Similarly, for port and role we need to define a new ViewElement (e.g, ViewEntityInterface). By extending our approach like this we can take into account metamodels for component-and-connector views as well. Clements et al. [5] defines several types of such views that include different types of components and connectors.

Our HOT builds a complete abstract syntax tree of an ATL transformation. Because we used a declarative style for defining it, and ATL does not allow navigation of generated target model elements, we have to build similar fragments

of this tree multiple times. This results in a lengthy HOT (> 1000 LOC). In the case of our MADL metamodel the generated transformation contains only 118 lines of ATL code. Although our transformation is generic and has to be defined only once, it would be interesting to investigate whether other strategies result in a more concise HOT, for instance, using a template-based approach.

6 Related Work

Our work can be understood as a model-driven approach to reflexion models [3]. To generate such a model, a high-level model is combined with a source model and a user-provided mapping between the two. The reflexion model indicates where source model and high-level model agree in terms of convergent, divergent and absent relations. Although our merged conformance model is clearly based on reflexion models, we also indicate conformance for relations. The flexibility of reflexion models for defining the mapping, makes them more suited for cases where the semantic gap between architecture and implementation is very large. Our approach could be improved by also allowing the user to provide a mapping. For this we need to modify our transformation such that it generates a comparison transformation that also takes a model of the user mapping as source model, which is then used in the `isMapping` helper (see Listing 1.3) for ViewElements.

Fabro and Valduriez [6] describe a tool to automatically match two metamodels. The aims of their work is to automatically generate a transformation that can transform the associated models into one another. Although their work is not aimed at conformance checking, the tool implements some useful heuristics to assess whether two metamodel elements match. We could reuse these in order to improve our `isMapping` helpers.

Other researched focused on the comparison of UML models. As an example, UMLdiff, an algorithm for calculating the difference between two UML Class diagrams, is presented in Xing and Stroulia [7]. The algorithm calculates the structural difference in terms of additions, removals, moves, and renamings of classes, packages, and their attributes and operations. Where UMLdiff enables fine-grained comparisons of UML class diagrams, our approach is not restricted to (the class model part of) the UML metamodel.

7 Conclusion

In this paper we outlined an approach for evaluating the conformance of software models using model transformations. Because such transformations are only generic with respect to a fixed metamodel for source and target models, we generalised our approach by defining a higher-order transformation (HOT) and a number of generic metamodel elements to be specialised for the definition of more specific metamodels. Based on a metamodel that uses these generic elements our HOT generates a transformation for checking the conformance of associated models. We demonstrated the application of our approach to a simple metamodel for the definition of architectural module-uses views.

Currently, we are in the process of improving and extending our approach. In Sect. 6 we already indicated how we could reuse other work for improving the identification of convergent, divergent, and absent entities and relations. Another potential improvement will be made by adding more flexibility. For some types of (architectural) views, with more complicated metamodels, not all types of elements are relevant for a conformance check. This might be the case when such elements are difficult to reconstruct from an implementation. We will add the possibility to ignore such elements.

Finally, we want to extent our approach also to *visualisation* of models. At the time of writing, we visualise models by transforming them to models that conforms to a metamodel for DOT, a graph description language. This metamodel is available from the ATL metamodel repository[1] and can be used to represent DOT graphs as models. From these models a grammar-based representation can be extracted and visualised using the *dotty* tool. The transformations to DOT contained many similar idioms with respect to how to visualise different types of elements. As such it makes sense to implement them as a HOT as well.

Acknowledgement Part of the research described in this paper was sponsored by NWO via the Jacquard Reconstructor project.

References

1. Hylke W. van Dijk, Bas Graaf, and Rob Boerman. On the systematic conformance check of software artefacts. In *Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005)*, volume 3047 of *Lecture Notes on Computer Science*, pages 203–221. Springer-Verlag, June 2005.
2. Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS2005*, 2005.
3. Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28. ACM Press, 1995.
4. Frédéric Jouault and Jean Bézivin. KM3: A DSL for Metamodel Specification. In *Proceedings of the 8th IFIP WG 6.1 International Conference Formal Methods for Open Object-Based Distributed Systems FMOODS 2006*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006.
5. Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures:Views and Beyond*. Addison-Wesley, 2002.
6. Marcos Didonet Del Fabro and Patrick Valduriez. Semi-automatic model integration using matching transformations and weaving models. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*, pages 963–970. ACM Press, 2007.
7. Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 54–65. ACM Press, 2005.

[1] `http://www.eclipse.org/gmt/am3/zoos/`