Comparing transformation languages for the implementation of certified model transformations

Arnaud Dieumegard, Andres Toom, Marc Pantel

Comparing transformation languages for the implementation of certified model transformations

Arnaud Dieumegard[1], Andres Toom[1,2], and Marc Pantel[1]

[1] IRIT - ENSEEIHT, Université de Toulouse, 2, rue Charles Camichel, 31071 Toulouse Cedex, France
FirstName.LastName@enseeiht.fr
[2] Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, EE-12618 Tallinn, Estonia
FirstName@krates.ee

Abstract. Precise specifications are needed for verifying and certifying the correct behavior of critical systems. However, traditional proofreading and test based verification techniques are usually not exhaustive and as systems become more complex, their coverage is less and less adequate. Use of models allows early verification, validation and automated building of "correct by construction" systems. Our work targets formal specification and verification of model transformations. In a previous paper we tackled the problem of writing formal specifications for model transformations independently to the implementation technique. In this paper we investigate the implementation phase of these specifications as model transformations using traditional MDE techniques and the difficulties encountered while generating the verification materials.

1 Introduction

Model Driven Engineering (MDE) is one of the techniques that has been successfully applied for designing complex systems. Splitting a system into layers and abstracting different system properties into respective models makes large systems manageable. In an iterative or V-like development process high-level system requirements and initial coarse models are refined until the requirements and models are precise enough to be implemented. In many cases, software code can be largely or fully automatically generated from low-level models. On the other hand, modeling languages have often (but not always) clearly defined syntax and semantics. This makes a model a formal specification analysable by formal mathematically based methods. Besides complexity, in critical embedded systems there is also a related strong concern regarding safety for the end-users and the environment. A critical software system cannot be released and embedded without complying fully with the certification of its corresponding domain, for example DO-178C in the avionics, ISO26262 in the automotive and ECSS for space systems. An important aspect in these normative guidelines is the need for clear separation between specification, implementation and verification. Splitting these concerns to separate tasks that can be allocated to independent parties (a requirement for highly critical systems) helps to eliminate both specification and implementation flaws. Most commonly the final implementation is verified against the low level requirements via extensive testing. If however, the requirements are in a formal language, then automatic verification or test generation can be applied. Ideally, the implementation can be even automatically generated from the requirement specification, as for example in the CompCert compiler project from Leroy [11] that is currently being experimented for flight control software by one of the authors [8]. This approach

was experimented in the GeneAuto project where this experiment also takes place by one of the authors [9]. However, the proof assistant technologies it relies on is costly and requires expert users that are currently not available in the software engineering industry that implements development tools for safety critical systems.

2 Case study: Verifying transformations in the GeneAuto code generator

GeneAuto[3] is an open code generator project for transforming a set of high-level graphical modeling languages to selected common textual programming languages (see [14, 10, 13, 1] that describe the evolution of the toolset in the last 6 years). It currently supports subsets of Simulink, Stateflow and Scicos as input and C, Ada and Java language as output. It is intended to be used and certified for critical embedded systems. That is why its design follows a clear modular MDE approach allowing to independently verify different transformation phases. After the initial importing step, transformations are carried out as a sequence of refinements of intermediate models.

There are about 50 transformations in the GeneAuto tool. Some of them are rather small and simple structure preserving transformations, but others are complex or change significantly the model structure. Related transformations are combined into independent executables that read and output the models to files. The transformation specifications are written with respect to these observable intermediate models. However, often these are a result of several successive transformations and verifying the structural correspondence between the input and output models is non-trivial. Here, explicit transformation links provided by the transformation tool can be very helpful, as shown by the next example.

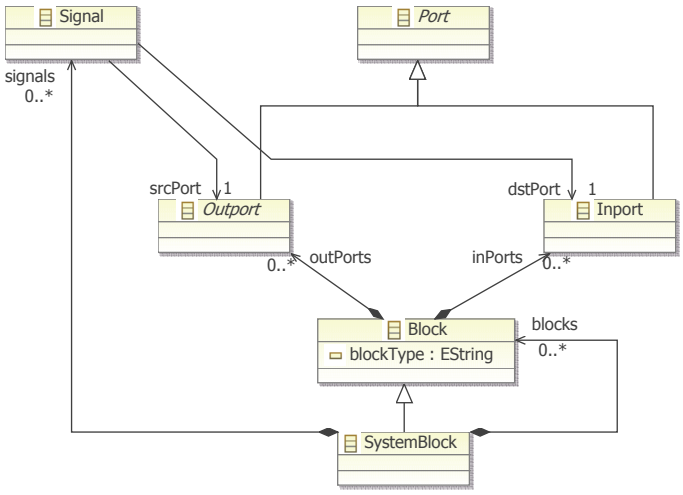

Fig. 1. A fragment of the simplified GASystemModel metamodel

[3] www.geneauto.org

Currently, the specification for most of the elementary tools in GeneAuto have been specified in the English language, with a notable exception of the Block Sequencer tool that has been specified and implemented in the Coq proof assistant [9]. In our case study we have formalized some of these requirements in a way that the specification can be directly used for transformation verification using a standard OCL checker. We will look at some transformations done by a tool called Functional Model Pre-Processor (FMPreProcessor), which handles normalizing and refinement of block diagrams. Figure 1 shows a section of the simplified GASystemModel metamodel with the relevant concepts.

The FMPreProcessor tool performs several quite simple transformations. Some of these are:

- Replaces system blocks that have corresponding library equivalents by library blocks
- Flattens virtual subsystems
- Matches and replaces primitive blocks with library blocks
- Removes Goto-From block pairs
- Determines the execution priorities of concurrent blocks based on their graphical position

As a concrete example we will look at removing Goto-From block pairs. Goto-From blocks (see Figure 2) allow to avoid visual clutter in block diagrams and split signals to sections. The GeneAuto tool removes such block pairs during the model preprocessing. A matching Goto-From pair is deleted, the endpoint of the first signal is moved to the endpoint of the second signal and the second signal is also deleted.

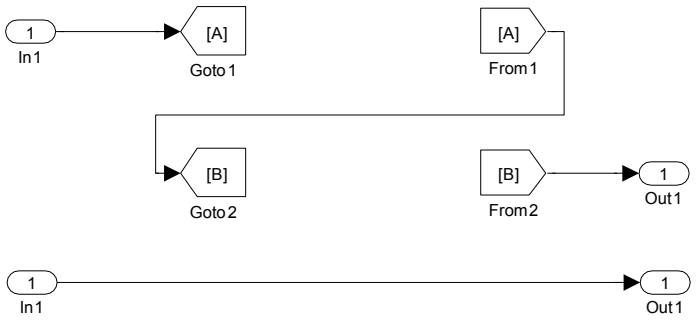

Fig. 2. Simulink diagram with chained Goto-From blocks before the normalizing transformation (above) and after it (below)

Verifying the correctness of this transformation is easy, provided that this is the only transformation. However, if transformations add up, even repetitive applications of the same transformation, like on figure 2, where the endpoint of the signal from From1 to Goto2 is also removed, then it will be much harder to verify the transformations correctness. Such analysis would have to determine in the source model the whole flow path from the block In1 to the first block that will not be removed. It would have to know a lot more about the model and transformation semantics. On the other hand, if the transformation tool maintains a link relating each port to a port in a target model, then a property like the correctness of a Goto-From removal can be specified and verified with a few simple OCL rules. All that the transformation tool has to do to allow it, is to store that after the first transformation:

1. Goto1, From1 and s2 were replaced by s1
2. The port corresponding to the input port of Goto1 is now the input port of Goto2

And after the second transformation:

1. Goto1, From1 and s2 were replaced by s1
2. The port corresponding to the input port of Goto1 is now the input port of Out1
3. Goto2, From2 and s3 were replaced by s1
4. The port corresponding to the input port of Goto2 is now the input port of Out1.

3 Metamodel based transformation specification

In the context of MDE the manipulated artefacts are models. The type of models is usually referred to as a metamodel: a model that defines the concepts of an instance model. This relation is purely syntactical. The Meta-Object Facility (MOF)[4] OMG standard defines a formal four-layered (M0..M3) metamodeling architecture. For example, the metamodel of the widely used Unified Modeling Language (UML) is a M2 MOF model. Similarly, all domain specific languages can also be expressed as MOF models. The core of MOF allows only expressing simple structural properties, like associations between elements, containment, cardinality etc.

The Object Constraints Language (OCL)[5] is a standard declarative first order constraint and query language, which can be used to refine MOF models. For instance, one can specify structural invariants, definitions and pre-post contracts of abstract MOF operations in the OCL language. Model transformations can also be specified as models and transformation constraints as correctness properties of the transformation model. These metamodels consist of a basic structural specification with possibly additional OCL constraints.

The transformation metamodel contains three essential parts: source, target and a transformation relation. The transformation relation is a set of explicit links between elements in the source and target model. These explicit links play a key role in our approach. These links must be explicitly given along with the transformation instance to allow feasible verification of the correctness of the transformation. In our approach these links are part of the specification and it is the responsibility of the transformation performer, be it a tool or even human, to provide these links. The relation metamodel defines the structure of the links and the properties they must satisfy and the transformation metamodel defines, which links must exist.

4 Transformation Metamodel

To perform model-based specification and verification of the transformation, we express the transformation of interest also as a model. Figure 3 shows our metamodel of the transformation model. This model has three main parts: references to source and target models and the relation links. The links refer to some elements in the source and target models and we expect the transformation tool to provide them.

A valid transformation model needs to comply with the transformation metamodel and a set of additional OCL constraints. Some of these constraints just specify the basic consistency of a

[4] http://www.omg.org/mof
[5] http://www.omg.org/spec/OCL/2.2

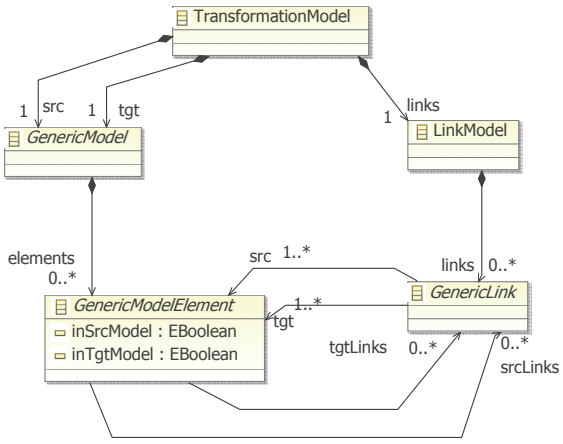

Fig. 3. Transformation Metamodel

transformation model. For example, to ensure the correctness of the relation links, we define the following OCL constraints (represented partially, the OCL keyword *inv* defines an invariant and the keyword *context* specifies, for which class the constraint applies to).

```
context GenericModelElement
inv src_has_only_src_links : inSrcModel
      implies tgtLinks->isEmpty()
inv tgt_has_only_tgt_links : inTgtModel
      implies srcLinks->isEmpty()
inv src_links_start_from_src : ...
inv tgt_links_end_in_tgt : ...

context GenericLink
inv src_elems_all_defined    : ...
inv tgt_elems_all_defined    : ...
```

Other constraints are transformation specific and are specified in either the context of the respective model elements or the relation links. We use a following general pattern: for source and target metamodel elements we specify as class invariants which links they must have and for links we specify as class invariants the respective transformation's correctness properties.

5 Implementing the specified model transformation

There exists a wide quantity of model transformation techniques that can be used to fulfill this purpose. We will describe in this section what are the main techniques that can be used in order to encode these transformations with languages that are accessible to common engineers. We will focus on the use of MOF-based[6] model transformations techniques.

[6] `http://www.omg.org/spec/MOF/`

5.1 Model to model transformations

In [6], the authors propose a classification of model transformation approaches:

- Direct manipulation of the models. Using some API to manipulate the model, the model transformer need to implement all the necessary methods and techniques to do the transformation.
- Relational approaches. Defining relations between source and target elements and specifying the constraints that must be verified by the relation.
- Graph transformation based approaches. This approach is based on the graph transformation theories. Some LHS graph patterns are transformed in RHS graph patterns.
- Structure driven approaches. First the hierarchical structure of the target model is created and then the attributes and references are set on the target model.
- Hybrid approaches. This approach combines techniques from the previous categories.

Within the Eclipse toolset, the EMF[7] framework has been developed in order to provide a Java API to manipulate models. The model transformations used in eclipse are based on this API. The Ecore metamodel can be seen as the EMF version of the MOF standard (more precisely the EMOF standard). Based on the EMF framework it is now possible to implement a direct manipulation of models.

Using directly a programming language like Java to implement a model transformation is the most obvious way to perform the transformation. This language is widely used and known by engineers. The input model of the transformation must be loaded and methods must be implemented in order to do the transformation. But this approach has the main drawback of being painful to implement as it is mostly mandatory to rewrite every mechanism for each implementation of a transformation.

OMG has defined a standard for model transformations: QVT[8]. This work started in 2002 and the first version of this standard was released in 2008. In the QVT standard, three metamodels are defined in order to provide declarative (with the Relation and Core languages) and imperative (with the Operational language) ways of performing the desired transformation. This hybrid approach allows users to implement simple transformations using the declarative languages and more complex ones using the imperative language.

5.2 Implementation

Our purpose in the previous sections and paper was to express how a formal model transformation specification can be written based on a transformation metamodel and OCL constraints. Here we will detail some experiments we did on the implementation of the Goto-From transformation specified in the section 2. The complete source code of the examples can be found at `http://dieumegard.perso.enseeiht.fr/implementation`.

In order to implement the transformation described in section 2 we need to follow the following algorithm:

1. The unmodified elements must be translated on the target model without modifications
   - all the blocks that are not Goto or From blocks (with their ports)
   - all the signals that are not involved in Goto-From pairs

[7] `http://www.eclipse.org/modeling/emf/`
[8] Query/View/Transformation: `http://www.omg.org/spec/QVT/`

2. A link must be created between each translated element of the first step.
3. Foreach signal leaving a not From block and going to a Goto block, a recursive treatment must be done to implement the Goto-From pairs elimination.
4. A link must be created on each iteration of the third step. This link contains as source all the signals and blocks involved in the Goto-From pairs and as target the resulting signal of the transformation.

Java/EMF based implementation Our Java based experiment is a slight retroactive extension of the existing FMPreProcessor component of the GeneAuto toolset. As explained in Section 2 most transformations in GeneAuto are endogenous (same source and target metamodels) model refinements in one of the two intermediate languages. The metamodel (abstract syntax) of these languages has been specified in UML and the skeleton Java classes have been automatically generated. Most transformations are manually coded. In our case study we have used the GeneAuto-EMF bridge and built a transformation metamodel on top of the GeneAuto metamodels. This transformation metamodel is the same one that was used also in the other experiments described in the current paper.

We augmented the existing transformation with the creation of explicit relation links between selected source and target model elements. This change was unintrusive and only made some internal information externally observable. To manage the links we employed a following strategy. First, the tool creates a partial identity relation (all elements are mapped to themselves) between selected elements (all instances of certain element classes) of the input model. Then, the tool performs an in-place transformation of the model and each time, when an element is replaced by another one in the course of the refinement or the model structure is otherwise changed the respective links are also modified. For example, if a Goto-From block pair is removed from the model a GotoFrom2SignalLink is added to the link relation and existing Port2PortLinks are updated. The fact that in our transformation metamodel the relation links and model elements are doubly linked makes this kind of updates very simple and efficient.

Such approach can also be extended to exogenous (different source and target metamodels) model transformations. In that case the initial identity relation creation phase is not relevant and the tool just has to output target model along with the the relation links.

QVT-Relational implementation In order to implement the transformation in QVTr we use the medini-QVT [9] tool set. There are many possibilities in QVTr to implement this transformation. The target and links of the transformation can be generated directly using a single transformation but it is necessary to launch the transformation in direction of the target model then in the direction of the link model to ensure that every element is created. An other possibility is to generate only the target element and here there are two ways of generating the transformation model:

– As the tooling generates automatically a trace model and metamodel for the transformation, an other transformation can be written in order to translate this automatic trace into a trace as specified in the section 4. This second solution has a major drawback because automatically generated traces are quite verbose and relies on the structure of the first QVTr transformation code (name of the relations and attributes). This implementation is not easily done and difficultly maintainable.

[9] `http://projects.ikv.de/qvt/`

– A transformation can be written taking as input the source and target models and generating the transformation model. This solution is easier to implement but a proofreading must be done between the first and second transformation in order to ensure that the links elements are created according to the first transformation.

We choose to implement the first version because of the drawbacks specified below.

The first part of the algorithm is quite easy to implement using the pattern recognition mechanism of QVTr. A mapping relation is written for each type of elements that must be directly translated to the target model. A specific relation is created to generate the GenericLink related to the translation and called directly on the where clause of the mapping relation. In order to implement the second part (the Goto-From transformation), a relation matching the entry point is written. A call to an other relation witch does the recursion over the Signals and Blocks and a final relation is used to create the resulting signal. During all this process, the Signals and Blocks used must be stored in order to fill the GenericLink for the trace. The same mechanism is used to call the trace creation relation but this time with the stored Blocks and Signals.

ATL implementation As ATL is a QVT-like transformation language, the principle of the transformation is the same as previously stated. The particularity of ATL is the possibility to add imperative treatments in the rules (relation in QVTr) at the end of each transformation rule. This eases the coding work but on the other hand does not helps on the readability of the transformation code. A lot of treatments needs to be done using helpers and the usage of filter in the pattern matching for rules is necessary to ensure that no mistakes are made on the application of the rules in the first or the second part of the algorithm. With ATL, the transformation is direct and it's possible to directly generate the target model and the links model using a unique execution flow.

6 Advantages and drawbacks of the experimented model transformations techniques

In this section we will discuss on the different implementations of the model transformations we describe on the previous section. We will base our comparison according to three criteria.

– Ease in generating the links model. Is it difficult to add a generation of the links model in addition to the target model.
– Expressiveness. What are the limits of the approach to express model transformations?
– Performance. What is the performance of the transformation?

6.1 Ease of implementing the transformation

Java/EMF Using either an UML or Ecore metamodel, the EMF toolkit can be used to generate the java classes representing the model elements. This capability is available also in most UML modeling tools. The transformation itself needs to be implemented manually. The main drawback of this approach is the amount of required manual coding. Every mapping mechanism or pattern recognition must be implemented directly using the Java language. In a larger project like GeneAuto it is likely that some kind of a framework or set of utility functions will be created first and then used for coding several transformations.

Creating the explicit relation links that we propose to use for verification does not add much overhead. The transformation tool only needs to keep and maintain some internal information that would otherwise be discarded. It is of course best, if the exact requirements for these links are specified in full detail before implementing the transformation. However, if the tool is well-structured it should not be complicated to add this also retroactively, as we did in our experiment.

QVTr The version we choose to implement in the section 5 is quite easy to implement. The mapping mechanism of QVTr is well suited for this usage. Every time a direct mapping is done, a relation creating the links is called in its where clause in order to create the link related to the source and target elements of the relation. The transitive closure aspect of our transformation requires to use the OCL conditional statements and a recursive relation but the QVTr language is well fitted for this usage. Some tweaks are necessary to store the Signals and Blocks during the transitive closure step of the transformation and use them to generate the links in a dedicated relation.

ATL Writing this transformation in ATL is quite easy and as we said on the previous section the imperative part of an ATL rule helps a lot to manage the creation of the elements and the flow of execution. It's also possible to generate the target and link models directly in one execution so it makes the rule writing more intuitive.

6.2 Expressiveness

Java/EMF Using an API in a full programming language like Java brings the expressiveness of that language to the transformation implementation. This makes any computable transformation possible.

QVTr By its nature, QVTr is limited to relations between source and target model elements. QVTr allows to express high level representation of model to model transformations. But it is sometimes difficult to do complex transformations in a declarative style as used in QVTr.

ATL As we previously said, ATL is an implantation of the QVT proposal, but it is a hybrid of declarative and imperative programming. This improves its expressiveness and grants it with the ability to express any kind of transformations but contrary to QVT, ATL transformations are unidirectional.

6.3 Performance

Performance evaluations have been done on an average computer with various generated models containing from a hundred of elements to nearly ten thousand. Two kinds of models have been used, casual ones containing a low common rate of Goto-From pairs in sequence and unusual ones with higher rate of these elements (about half of the model elements are involved in Goto-From sequences). The second kind of models are only there for evaluation of the computational performance of the transitive closure. The results of this benchmark are displayed in Table 1.

Java/EMF The Java/EMF implementation had the best performance in our study. This is rather expected from a hand made "optimal" implementation.

Model type	Elements in model	Technology(execution time in s)		
		QVTr	ATL	Java/EMF
Low rate	$\simeq$ 150	379,556	3,441	1,057
Low rate	$\simeq$ 1500	—	32,802	4,660
Low rate	$\simeq$ 15000	—	5189,391	261,552
High rate	$\simeq$ 150	430,508	3,699	1,059
High rate	$\simeq$ 1500	—	29,771	4,681
High rate	$\simeq$ 15000	—	2647,606	132,752

Table 1. Model transformation performance

QVTr The performance of the medini QVT implementation of QVTr is very low. It is possible to handle small models containing a few hundreds of elements but using it with bigger models results in huge execution times. The main reason for that is the pattern recognition mechanism and the fact that every relation must hold for the transformation to be successful. For every relation in the transformation, it's necessary to go over all the model to find every matching and make them hold.

ATL In QVTr, the calls from a relation to another is done via the where clause but in ATL, this is not the only way to call other rules. It is possible to specify in a rule that the specified mapped element must be transformed by transforming a specific element of the source model. Using this mechanism, it's possible to reduce the number of elements that needs to be matched in a transformation rule. The fact that ATL is also compiled and executed in a virtual machine makes the execution faster than classical interpretation of the transformation rules. Using ATL we experienced good transformation performance. However, the fact that we developed only small transformations is not irrelevant.

7 Perspectives

As we target this work to be used in the modeling community and in industrial contexts, we plan to extend it to other different transformations in order to test the scalability of the approach both in size and practical complexity.

We plan to continue the GeneAuto experiment and formalise a larger part of the transformation specification in the style proposed in Section 3, adapt parts of the existing GeneAuto tool to produce the required trace links and study the correctness of the model transformation and the feasibility of this verification approach on realistic models.

We also plan to address the verification of the specification itself by developing a methodology for combining the syntactic transformation specification with additional semantics-based analysis to show the soundness and completeness of the transformation specification with respect to the semantics of the source and target languages. By verifying the conservation of the semantic properties by the application of the transformation we may be able to prove the soundness of the whole transformation independently of the implementation language.

And last, we plan to experiment with other model transformation languages like Kermeta, QVTo and graph rewriting based transformation engines. These elements will be added on the `http://dieumegard.perso.enseeiht.fr/implementation` web page during the forthcoming weeks.

8 Related works

Many authors target the verification of model transformations or code generation with various purposes and technologies (see [7] for a compiler verification bibliography). The main specific aspects of our proposal are that: a) we target qualification with respect to certification standards; b) thus we must consider the global process including independent specification, implementation and verification activities; c) we rely on OMG standards for the specification and do not enforce any technology on the implementation; d) we target only structural properties and propose to handle semantic aspects using more appropriate technologies for the specification validation in a separate phase done by different people than the one that implement the transformation; and e) we must be able to handle industrial size models. As this paper mainly focuses on the implementation part of this approach, we will compare this to others present in the literature. We detailed in our previous paper a comparison of the other aspects of our approach.

Most of the related works, as [5, 4, 3] or [2] relies on the links inside the transformation model in order to ease the verification of transformation. They propose to extract the links from declarative transformation languages. However, these links are most of the time implementation links that appear each time a transformation rule is applied. Thus, either the rules are written in order to ease the implementation and you usually get much more links than needed, or you introduce constraints on the way you can implement the transformation in order to produce exactly the links needed for the verification. Also, there is no implementation aspect described in these proposals.

Narayanan et al. have also proposed something very similar to our proposal in [12]. They advocate to focus on structural properties, to specify the transformation as relations between the source and target metamodels and then to extract these links from the cross-links used for the implementation of the transformation with the GReaT language. There is a potential drawback if the structures of the source and target metamodels are very different. It might be required to build the transformation using several intermediate models as it is usually the case with declarative languages. Then it might get complicated to retrieve the specification link that is a composition of many implementation links.

The transitive closure of a transformation rule is already a complex case: should it be translated to a single specification link, to all the intermediate links, to only the implementation links? We enforce the implementation to build exactly the right links that must be precisely described in the specification. This introduces a cost on the implementation side but it also relieves the implementation team from the constraint of using a declarative transformation language. Moreover, as in our approach it is possible to reference the links in the transformation specification, complex and composed transformations can be specified and implemented more flexibly.

9 Conclusion

We have described in this paper a pragmatic approach for the implementation of formally specified model transformation and some examples to describe how the creation of the transformation model can be done. This approach is well suited to the development process of complex and critical software. Verification of the implementation with respect to the specification is performed automatically using an OCL checker.

We have shown through the implementation of the same transformation with Java/EMF, QVTr and ATL that on one hand, such a model transformation specification is implementable in quite

different ways, of which some can be applied on industrial size models, and on the other hand, the same automatic verification technique can be used on all of those different implementations.

Further testing of the scalability of the approach for multiple and more complex model transformations needs to be done in order to assert the usability for more complex use cases. We plan also to develop a methodology for combining the syntactic transformation specification with additional semantics-based analysis to show the soundness and completeness of the transformation specification with respect to the semantics of the source and target languages.

References

1. Bordin, M., Naks, T., Toom, A., Pantel, M.: Compilation of heterogeneous models: Motivations and challenges. In: European symposium on Real Time Software and Systems (ERTS^2), Toulouse, 29/01/08-01/02/08. p. (electronic medium). Société des Ingénieurs de l'Automobile, http://www.sia.fr (2012)
2. Büttner, F., Cabot, J., Gogolla, M.: On Validation of ATL Transformation Rules By Transformation Models. In: Cichos, H., Fondement, F., Lucio, L., Weissleder, S. (eds.) Proc. Workshop on Model-Driven Engineering, Verification, and Validation (MODEVVA'2011). IEEE (2011)
3. Cariou, E., Ballagny, C., Feugas, A., Barbier, F.: Contracts for model execution verification. In: France, R.B., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA. Lecture Notes in Computer Science, vol. 6698, pp. 3–18. Springer (2011)
4. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: Ocl contracts for the verification of model transformations. In: OCL workshop of MoDELS (oct 2009)
5. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: Ocl for the specification of model transformation contracts. In: Workshop OCL and Model Driven Engineering of the Seventh International Conference on UML Modeling Languages and Applications (UML 2004) (oct 2004)
6. Czarnecki, K., Helsen, S.: Classification of model transformation approaches (2003)
7. Dave, M.A.: Compiler verification: a bibliography. ACM SIGSOFT Software Engineering Notes 28(6), 2 (2003)
8. França, R.B., Favre-Felix, D., Leroy, X., Pantel, M., Souyris, J.: Towards formally verified optimizing compilation in flight control software. In: Lucas, P., Thiele, L., Triquet, B., Ungerer, T., Wilhelm, R. (eds.) PPES. OASICS, vol. 18, pp. 59–68. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2011)
9. Izerrouken, N., Pantel, M., Thirioux, X.: Machine-checked sequencer for critical embedded code generator. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 5885, pp. 521–540. Springer (2009)
10. Izerrouken, N., Thirioux, X., Pantel, M., Strecker, M.: Certifying an automated code generator using formal tools : Preliminary experiments in the geneauto project. In: European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008. p. (electronic medium). Société des Ingénieurs de l'Automobile, http://www.sia.fr (2008)
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
12. Narayanan, A., Karsai, G.: Verifying model transformations by structural correspondence. ECEASST 10 (2008)
13. Toom, A., Izerrouken, N., Naks, T., Pantel, M., Ssi-Yan-Kai, O.: Towards reliable code generation with an open tool: Evolutions of the gene-auto toolset. In: European symposium on Real Time Software and Systems (ERTS^2), Toulouse, 29/01/08-01/02/08. p. (electronic medium). Société des Ingénieurs de l'Automobile, http://www.sia.fr (2010)
14. Toom, A., Naks, T., Pantel, M., Gandriau, M., Wati, I.: Gene-auto - an automatic code generator for a safe subset of simulink-stateflow and scicos. In: European symposium on Real Time Systems (ERTS), Toulouse, 29/01/08-01/02/08. p. (electronic medium). Société des Ingénieurs de l'Automobile, http://www.sia.fr (2008)