Specification and Validation of Model Transformations for Certified Systems' Development

Andres Toom, Arnaud Dieumegard, Marc Pantel

Specification and Validation of Model Transformations for Certified Systems' Development [⋆]

Andres Toom[1,2], Arnaud Dieumegard[1], 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. Certifying critical systems requires very precise specifications and ability to verify each development step. However, proofreading and test based verification are usually not exhaustive and as systems get 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. Such techniques provide significantly higher confidence of correctness and can even reach exhaustiveness. In this paper, we rely on common model driven engineering techniques to allow common engineers to write these specifications and to conduct verification. We propose to use a simple transformation model for specifying the expected relation between the source and target models after the transformation. The source and target metamodels are extended with a traceability model that defines a set of links that must exist after the transformation and whose correctness is specified as OCL constraints.

1 Introduction

Software is playing an increasing role in all domains, including safety critical domains such as avionics, railway, nuclear energy or medicine, besides more traditional domains. It is well known that all these systems are becoming more and more complex. Model Driven Engineering (MDE) is one of the techniques that has been applied quite successfully for designing complex software 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, modelling 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. Most of these are process-oriented, requiring that every stage of the software

[⋆] This work has been partly funded by the ITEA2 project OPEES, by ERDF through the Estonian Center of Excellence in Computer Science (EXCS) and by the French and Estonian Science Foundations throught the Parrot program.

development needs to be verified and verifiable. The exact means for achieving these goals are usually left open and are up to the party seeking certification or qualification. 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 the low-level requirements are in a form of human language, then the respective testsuites must be first manually created and their conformance and coverage of the requirements must be separately shown. 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 [13] 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 [10]. 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.

The use cases of transforming computer data differ largely in size and complexity ranging, for example, from transforming a configuration table to another configuration table upto translating a general purpose programming language to assembly code. While verifying the compilation of traditional programming languages to lower level languages has been studied a lot and there exist many good techniques, the studies of Domain Specific Language (DSL) transformation, model transformation and code generation is much less established. This is largely due to the fact that each case is somewhat unique - developed according to a particular need, the languages are likely to change over time and moreover, there might not be a clear formal specification of the respective languages and a suitable semantics must be chosen during the specification of the transformation. Often the semantic gap between the source and target languages of such transformations is also larger than in traditional program compilation. The approach we present here is a lightweight syntactic-structural way to specify the transformations in the Model Driven Engineering (MDE) fashion as model transformations and specify the details of the transformation relations using Object Constraint Language (OCL) constraints. Our approach has several similarities with the ones in [15] and [4], but has an important methodological difference - we are making the links between the source and target models explicit and part of the transformation specification. Our approach is a gray-box approach, which has several advantages when dealing with complex transformations.

The paper proceeds as follows. We will first present a model transformation case study that is based on the GeneAuto embedded code generator in Section 2. We will then give a high-level overview of the proposed transformation specification and verification scheme in Sections 3 and 4. In Section 5 we will formalise the specification of the transformation used in the case study and in Section 6 we describe an experimental setup that implements the whole approach. In Section 7 we explain how some routine part of the specification can be automated and in Section 8 comment some possible extensions of the approach.

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 modelling languages to selected common textual programming languages (see [20, 11, 19, 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 and Ada language as output. It is intended to be used and certified for critical embedded systems. That‘s 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, as displayed on Figure 1.

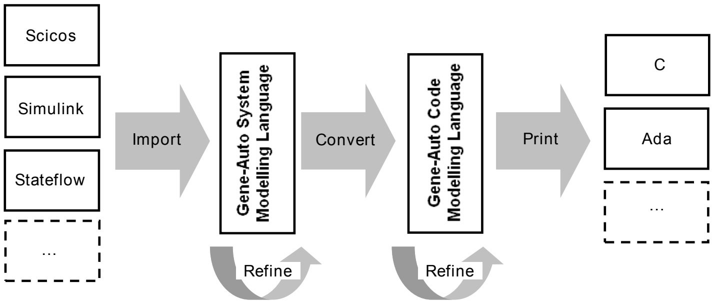
Scicos
Simulink
Stateflow
...
Import
Gene-Auto System Modelling Language
Refine
Convert
Gene-Auto Code Modelling Language
Refine
Print
C
Ada
...

Fig. 1. Model transformation chain in the GeneAuto code generator.

Altogether there are about 50-60 transformation passes in the tool depending on the configuration. Some of them are rather small and simple structure preserving transformations, but others are rather complex or change significantly the model structure. For practical purposes collections of transformations are combined into independent executables called elementary tools. An elementary tool reads its input model from a file and writes the output model to another file. The low-level tool requirements or transformation specification is 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. 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 [10]. The natural language requirements are of course incomplete, ambiguous and not directly verifiable. In our case study we have formalised some of these requirements in such a way that the specification can be directly used for transformation verification according to the scheme described in Section 3. We will look at some transformations done by a tool called Functional Model Pre-Processor (FMPreProcessor), which handles normalising and refinement of Simulink and Scicos block diagrams. The tool is executed right after importing a user model (a diagram). At this point there is only rather basic information

[3] www.geneauto.org

known about the model. Figure 2 shows a section of the simplified GASystemModel metamodel with the relevant concepts.

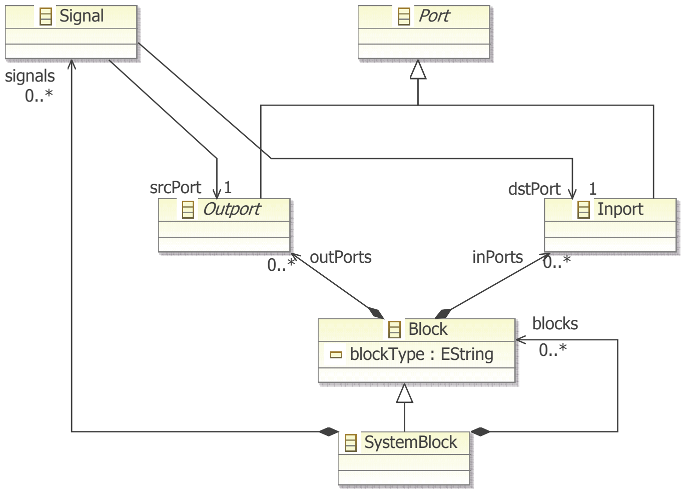

Fig. 2. A fragment of the simplified GASystemModel metamodel

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

- Replaces subsystems (hierarchical 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 3) are a way in a Simulink diagram to avoid visual clutter 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.

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 3, where the endpoint of the signal from From1 to Goto2 is also removed, then it will be much harder to verify the transformation's 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. What all the transformation tools has to do to allow it, is to store that after the first transformation:

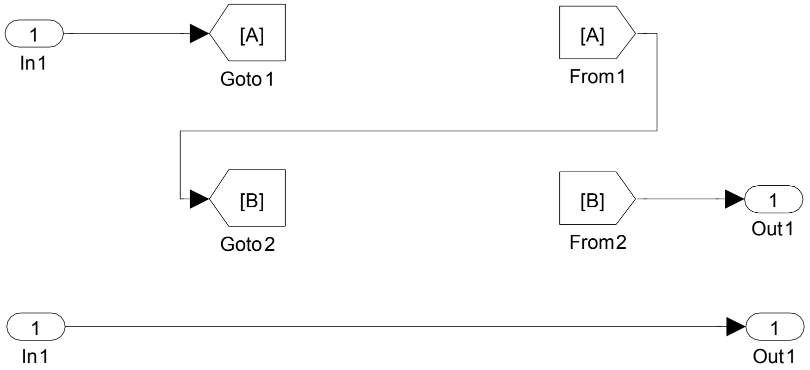

Fig. 3. Simulink diagram with chained Goto-From blocks before the normalizing transformation (above) and after it (below)

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 with explicit relation links

3.1 *A posteriori* transformation verification

To prove that an implementation of a program or model transformation (a transformation tool) matches its specification there are broadly two approaches. First, one could verify once and for all, that the implementation is correct for any possible input, as for example in [13]. This approach is naturally preferable, when possible. However, in some cases an alternative approach of verifying the correctness of the transformation *a posteriori* on a transformation instance level can be more feasible. This idea is not new. The approach and a term Translation Validation was proposed by A. Pnueli in [16]. The approach suggests that a program transformation phase should be followed by a verification phase, which independently from the transformation tool, verifies that the semantics of the source and target programs are the same. It further suggests that the transformation tool could provide hints to the verifier tool to help build the relation between the source and target programs more efficiently, but these should not affect the logical outcome of the verification phase. This approach generally has the obvious drawback of having to perform the verification after each transformation instance, when compared to the approach of a verified transformation tool. However, its main advantage is a simpler specification of the properties of interest and a simpler verification tool. Hence, the approach can be of advantage, when the requirements for the transformation tool are complex and evolving, as is often the case for realistic compilers or domain specific program transformers. Also, verifying the correctness of the verifier itself can be simpler in this case.

3.2 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) metamodelling 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. Figure 4 presents the high-level view of our approach. Both, the transformation input and output models are expected to conform to their respective metamodels. These metamodels consist of a basic structural specification with possibly additional OCL constraints. The input and output metamodels can be the same or different. The first kind of transformation is also known as a refinement or endogenous transformation and the second one as exogenous.

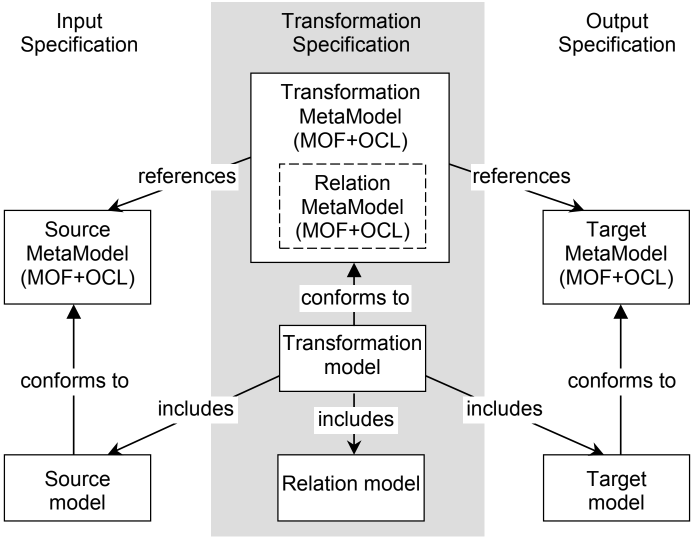

Fig. 4. High level view of the model transformation specification

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

[4] http://www.omg.org/mof

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

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. Such an approach is also called a gray-box approach: we require some information from the transformation performer, but we don't need to know all the details of the implementation. The number and kinds of links that must be specified is transformation specific and somewhat open. The relation metamodel defines the structure of the links and the properties they must satisfy and the transformation metamodel defines, which links must exist.

3.3 Checking the correctness of the transformation

In this setting, the correctness of the transformation instance can be simply verified by checking first, that the target model conforms to the target metamodel and its constraints (conformance of the source model to its metamodel and constraints can be assumed and is likely to be at least partly verified during the transformation phase), that the transformation model structurally conforms to the transformation metamodel, that all the required translation links exist and finally that all links satisfy the respective constraints. These checks can be carried out using any standard metamodelling framework and an OCL checker. Figure 5 displays a high-level view of such a workflow.

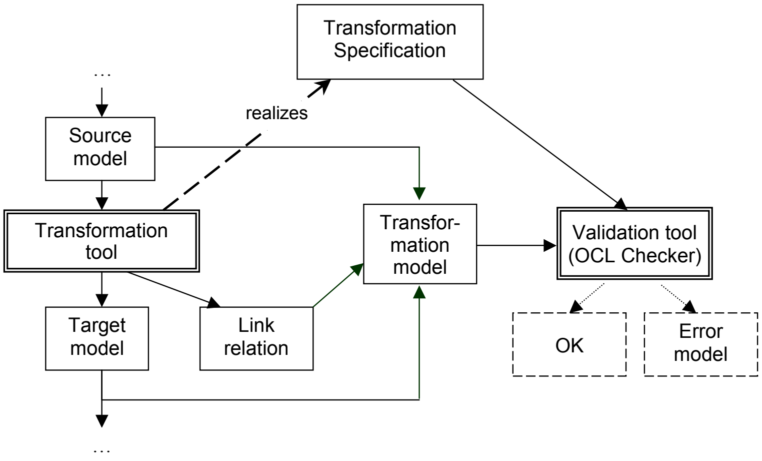

Fig. 5. High-level view of model transformation and verification

Structural correspondence verification of a transformation requires relating and element-wise comparing of elements in source and target models. The main difficulty relies in matching the corresponding elements. If the transformation is simple and structure preserving, then establishing the structural correspondence relation is straightforward and can be even automated [5, 4, 15, 14]. However, if transformations are more complex or combined, then this correspondence is much harder to establish, resulting in complex and computationally expensive verification criteria. An example is given in Section 2. Explicit transformation links, on the other hand, make it trivial to identify the related elements. It is also not necessary for all elements of source and target models to be explicitly linked. Generally, it is only required that the explicit relation is given for elements, where the location of the translated elements in the target model doesn't follow directly from the relative location of the source elements in the source model. This is also shown in Section 2.

Such an implementation can be used for early testing of the requirements and be replaced with a different one when needed. An important additional property of our approach is that: the transformation specification does not need to be complete to be usable. It can initially only address the parts whose verification is of primary concern and other parts can be specified as needed.

3.4 About executable specification

It must be noted that technologically this approach has some common points with an ”executable specification” approach. In its MDE version, the source and target specifications are also given by metamodels. The transformation specification too follows a certain metamodel : the metamodel of the transformation language (for example, QVT[6], ATL[7]). But differently from ours, the transformation model is the transformation specification and is executed with respect to a given source model to produce the target model. This approach might be good for some transformations. Especially, when the transformation can be specified declaratively. However, complex transformations are hard to represent purely declaratively. Secondly, such a transformation scheme might not satisfy the performance or other non-functional requirements that can be met in a custom transformation implementation. Thirdly, verifying the correctness of a generic model transformation engine is generally more complex than verifying an OCL checker. Finally, this approach alone does not answer the required separation of concerns and independence between specification, implementation and verification described in Section 1.

But on the other hand, since our approach presented above makes no assumptions of the transformation implementation technology, the “executable specification” can be treated as a realisation of the basic transformation specification and it can benefit from the source and target metamodel specifications. The only additional requirement is that besides the output model it must also produce the required explicit transformation links.

3.5 Interpretation of the verification result

The transformation implementation might not be perfect. If the verification passes for a specific transformation run, the output is known to satisfy the specified properties. If the verification fails, one has either the option to fix the transformation tool or the output. For large and complex applications fixing a bug might be a lengthy process and/or not under the control of the user of the tool. In some cases it might be possible to correct the output instead. If the verification then succeeds, the user has a verified output with similar guarantees as in the first case. Finally, a failure in the verification phase islikely to give messages that are understandable to the user. This is because the verification checks only structural properties of the source and target model and is able to immediately point the user to a violated rule in the OCL language and particular source and target elements. The OCL language is understandable by common engineers and there is no reference to intermediate data that has been built by the verification tools, only to the source and target models.

[6] http://www.omg.org/spec/QVT/1.0

[7] http://eclipse.org/atl

4 Transformation Metamodel

To perform model-based specification and verification of the transformation, we express the transformation of interest also as a model. Figure 6 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. For convenient specification and verification of completness properties we also use backlinks from model elements to links. However, these backlinks can be built automatically during the verification phase. To simplify the model transformation constraints, we have introduced an attribute inSrcModel and an inverse derived attribute inTgtModel to the GenerciModelElement class.

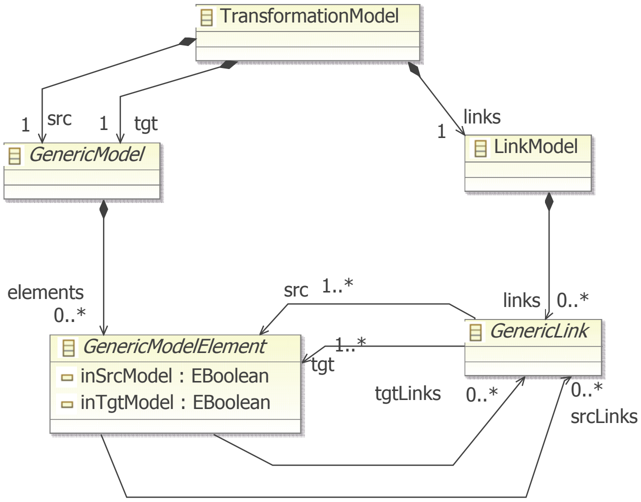

Fig. 6. Transformation Metamodel

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 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. This will be illustrated by an example in Section 5.

5 Correctness of the Goto-From elimination transformation

To specify the correctness of the Goto-From elimination described in Section 2 we define a specific link class GotoFrom2SignalLink. In the transformation specifiction we also refer to another link class Port2PortLink, which keeps track of the transformation of the port. This ability to refer and navigate through links in the transformation specification makes writing precise and directly verifyable specifications very convenient - we can refer to the image of any transformed element in the target model by following the link. This allows to specify a complex transformation compositionally and verify multiple transformations with one pass.

The main OCL rules that check the correctness of a Goto-From pair elimination are given below. Notice the getTgtPort operation used in the gotoTagCheck that navigates from a port of a source model block to a corresponding port in the target model. The target port, as explained in Section 2, might not be in the same relative location as the original.

```
context Block
inv Goto_block_has_GotoFromLink :
        type = 'Goto' implies srcLinks->one(
              oclIsKindOf(GotoFrom2SignalLink))
inv From_block_has_GotoFromLink :
        type = 'From' implies srcLinks->one(
              oclIsKindOf(GotoFrom2SignalLink))
inv Goto_block_inSrcModel_only :
        type = 'Goto' implies inSrcModel
inv From_block_inSrcModel_only :
        type = 'From' implies inSrcModel

context GotoFrom2SignalLink
inv sigStartTransf :
        srcGotoSig.srcPort.getTgtPort = tgtSig.srcPort
inv sigEndTransf :
        srcFromSig.dstPort.getTgtPort = tgtSig.dstPort
inv gotoTagCheck :
        srcGotoBlock.getParamValue('GotoTag') =
         srcFromBlock.getParamValue('GotoTag')
```

6 Experimental verification infrastructure

Our experimental verification framework has been written in Java using standard components from the Eclipse Modeling Framework (EMF[8]) for model handling and OCL checking. However, the approach is general and only requires capabilities to read MOF compliant models and execute OCL queries. The transformation tool that we use in our case study, FMPreProcessor, has also been implemented in Java, however, this is again not a restriction, since it is only required from the transformation tool to input and output MOF compliant models and output also the required link information relation as a MOF compliant model.

The transformation workflow using the GeneAuto toolset starts by either the SimulinkImporter or ScicosImporter tool that transforms a user model to an intermediate model SM_1 in the GeneAuto System Model Language (GASystemModel). This model is read by the FMPreProcessor tool, which transforms it to SM_2 and outputs also the required links. The SM_2 model is in turn read by the next tool in the transformation chain.

The verification workflow begins with a small tool called TransformationBuilder that reads the input and output models and links, and converts them to a Transformation Model. This tool also builds the backlinks described in Section 4. Alternatively, the Transformation Model could be also built by the transformation tool, but then one must additionally verify that the source part of the Transformation Model exactly corresponds to the original source model of the transformation (SM_1, in our example), as the Transformation Model was built by the tool we are verifying and we should not trust it. The Transformation Model is then fed to an OCLChecker tool along with a set of OCL files. The OCLChecker processes the OCL files one-by-one and checks the constraints in it with respect to the relevant elements in its input model (the Transformation Model). It produces as output either the result "Success" or an Error Model containing information about the failed constraints and references to concrete model elements that violated them.

Preferably, the formal specification exists before a transformation is implemented. However, it is possible also to refine the specification and apply the presented approach at a later phase. In our case study the FMPreProcessor tool already existed, when carrying out the study. We made small non-intrusive modifications in the tool to record and output the relation links. If the requirement to maintain the transformation links had existed beforehand, it would have cost no additional effort at this stage. Additionally, we have implemented an interface between the presented verification toolchain and the existing GeneAuto testsuite enabling to run the verification automatically on a large number of models. We plan to extend this study to specify and verify a larger part of GeneAuto in this style. The material related to the experiment can be found at `http://toom.perso.enseeiht.fr/verification`.

7 Automating specification of refinement transformations

In refinement transformations the source and target metamodels (languages) are the same. Such transformations have significant practical importance. For instance, in the GeneAuto embedded code generator that we use in our case study there are about 50-60 model transformation phases, but there are only two intermediate languages and the majority of the transformations are endogenous.

Endogenous transformations change only a part of the model and one also needs to specify that the "other parts" do not change. This can be a rather tedious task to do manually for large

[8] `http://www.eclipse.org/modeling/emf`

metamodels. However, it can be partly automated. It is relatively easy to generate the specification of an identity transformation. Then the specification can be manually refined to take into account the required changes in the model. For instance, ATL can be used to generate a part of the relation metamodel and the respective OCL constraints based on the source metamodel. We have partly used this in our case study.

8 Perspectives

As we target this work to be used in the modeling community and in an industrial context, we plan to extend it on other transformations in order to test for the scalability of the approach both in size and practical complexity. For the moment it's difficult to measure the difficulty to write the full specification of a code generator like Geneauto without proceeding to further investigations. The practical complexity must also be addressed by describing a taxonomy of transformations constraints adapted to model transformations specification.

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.

9 Related works

Many authors target the verification of model transformations or code generation with various purposes and technologies (see [6] for a compiler verification bibliography). We restricted this comparison to the closest ones: the use of model driven engineering technologies for the specification and verification of model transformation using the translation validation method. 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.

Many proposals rely on the use of formal methods and targets both structural and semantics issues. Based on the feedbacks from the industrial partners in the GeneAuto project, we have chosen to focus on structural properties in order to ease the use of our proposal by classical software engineering team in the industry and its acceptation by certification authorities. Semantics issues will be handled at the specification validation phase by specialists. On the structural properties side, declarative languages based on rewriting rules have been the subject of many proposals based on model checking technologies (see Varro et al. [17]) or rewriting technologies like concluence or termination checking (see Taentzer et al. [9, 12, 7]). However, these technologies do not scale well to industrial size models.

Cariou et al. have proposed in [5] to specify model transformation using pre and post conditions on the source and target metamodel expressed using OCL constraints. This work was improved in [4] and experimented on another use case in [3]. The key difference is that they propose to build the transformation model automatically based on the available information in the source and target models. The implementation can thus be a black-box one. However, if the metamodels do not contain the appropriate information, it might not be possible to build the mandatory links, or it might be very costly for industrial size models. The use of explicit traceability links through the transformation model allows to alleviate that risk at the cost of enforcing the transformation to build the links. However, this kind of links are already mandatory for the development of certified systems. Thus, they do not increase the costs.

Narayanan et al. first proposed in [14] to use translation validation for verifying the correctness of model transformation. They focus on the verification method and rely on the cross-links created as part of the the transformation in the GReAT[9] language much in the same manner as the traceability links from the transformation model in our proposal. Their proposal could be applied to most of the declarative transformation languages that enforce the use of explicit or implicit links between source and targets in order to execute the transformations, e.g. the QVT/Relational standard, Atlas Transformation Language (ATL[10]) or Triple Graph Grammar [18] based tools like Fujaba[11] or Moflon[12], ... 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. This last case is sometime not possible as the links are also needed for the internal management of the transformation. This work also focus on semantics verification that is significantly more complex and cannot usually be handled by common software engineers.

Büttner et al. have proposed recently in [2] to rely 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 taking ATL as use case and then implement verification activities as OCL constraints on the extracted model. The verification approach is very close to our proposal. However, as the links are derived from the transformation implementation, the drawbacks stated in the previous paragraph still apply.

Narayanan et al. have also proposed something very similar to our proposal in [15]. 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. Morevover, as in our approach it is possible to reference the links in the transforma-

[9] http://repo.isis.vanderbilt.edu/tools/get_tool?GReAT

[10] http://eclipse.org/atl

[11] http://www.fujaba.de

[12] http://www.moflon.org

tion specification, complex and composed transformations can be specified and implemented more flexibly.

10 Conclusion

We have described a pragmatic approach for specifying and verifying model transformations using standard modelling techniques, explicit specification of a subset of the transformation relation and OCL for specifying the transformation's correctness properties. Such an approach is well suited to the development process of complex and critical software. It promotes early formalization of the system's requirements and supports a separation of concerns between the domain specialist, who specifies the initial requirements, implementor of the transformation and a verifier. Verification of the implementation with respect to the specification is performed automatically using an OCL checker. However, the specification itself can be further verified by a verification specialist, who can independently verify the soundness and consistency of the specification by formal techniques, e.g. theorem proving. The last step is facilitated by the fact that the requirements have already been specified in a formal language (MOF with OCL). On the other hand, the transformation specification does not need to be complete to be usable, thus allowing to verify some properties of a transformation specification and implementation already at a very early stage.

On a high-level, the proposed approach is similar to the Translation Validation technique [16]. This has, on one hand, the obvious drawback of having to perform verification after each transformation instance. However, it has also advantages, when the transformation specification is complex and subject to evolvement, as in many realistic tools. The fact that in our case the specification is expressed strucutrally and using standard modelling techniques, makes the approach applicable also in an industrial context by industrial engineers. In a certified/qualified context it is also very important to be able to feasibly vertify the verification toolchain itself. This is facilitated by the fact that our approach makes use of a rather lightweight toolchain and standard components.

We have developed an experimental verification framework for testing our approach using Java and components from the EMF framework. We have formalised the transformation specification of some of the transformations from the GeneAuto embedded code generator and verified their correctness using the scheme proposed here. This experiment has been integrated in the GeneAuto nightly built testing framework. We plan to extend this work to other transformations in order to test the scalability of the approach both in size and practical complexity. 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. Dave, M.A.: Compiler verification: a bibliography. ACM SIGSOFT Software Engineering Notes 28(6), 2 (2003)
7. Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination criteria for model transformation. In: Cerioli, M. (ed.) FASE. Lecture Notes in Computer Science, vol. 3442, pp. 49–63. Springer (2005)
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. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G. (eds.) ICGT. Lecture Notes in Computer Science, vol. 2505, pp. 161–176. Springer (2002)
10. 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)
11. 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)
12. de Lara, J., Taentzer, G.: Automated model transformation and its validation using atom 3 and agg. In: Blackwell, A.F., Marriott, K., Shimojima, A. (eds.) Diagrams. Lecture Notes in Computer Science, vol. 2980, pp. 182–198. Springer (2004)
13. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
14. Narayanan, A., Karsai, G.: Towards verifying model transformations. Electr. Notes Theor. Comput. Sci. 211, 191–200 (2008)
15. Narayanan, A., Karsai, G.: Verifying model transformations by structural correspondence. ECEASST 10 (2008)
16. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS. Lecture Notes in Computer Science, vol. 1384, pp. 151–166. Springer (1998)
17. Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: A comparison of two approaches. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT. Lecture Notes in Computer Science, vol. 3256, pp. 226–241. Springer (2004)
18. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT. Lecture Notes in Computer Science, vol. 5214, pp. 411–425. Springer (2008)
19. 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)
20. 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)