Specifying and Verifying Model Transformations for Certified Systems using Transformation Models

Andres Toom, Arnaud Dieumegard, Marc Pantel

Specifying and Verifying Model Transformations for Certified Systems using Transformation Models

Andres Toom[1,2,3], Arnaud Dieumegard[1] and Marc Pantel[1]

***Abstract*— Model Driven Engineering (MDE) makes it possible to tackle the ever-rising complexity of developing safety critical systems by providing early verification & validation (V & V) and automated model transformations. In order to have confidence in them, certification guidelines require full traceability between the transformation's source and target and V & V to prove the correctness of the transformation and verification tools themselves. This work was carried out as a separate study based on the GENEAUTO[1] project for the development of an automatic code generator for certified systems. It focuses on the use of the MDE *"Transformation Model"* approach from [1] as a compromise between the current industrial practice, trends in software engineering, safety standards and formal verification. It outlines the key points for the development of model transformations for certified systems and illustrates the use of a form of the *"Transformation Model"* on a preprocessing step in GENEAUTO.**

***Index Terms*— MDE, transformation model, embedded systems, certification, qualification, MOF, OCL**

I. INTRODUCTION

Model Driven Engineering (MDE) is a technique that has been applied successfully in the design of complex systems. In the usual development process the high-level system requirements and initial coarse models are refined and formalized as models, until they are precise enough to be implemented. In many cases, software code can be largely or fully automatically generated from the low-level models.

This work has been funded by the FUI Project P, EuroStars project Hi-MoCo and the Estonian Ministry of Education and Research target-financed research theme No. 0140007s12.

[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

[3]IB Krates OU, Maealuse 4, EE-12618 Tallinn, Estonia FirstName@krates.ee

[1]http://www.geneauto.org

While verifying the compilation of traditional programming languages has been studied a lot, the fields of Domain Specific Languages (DSL), model transformations and code generation are much less established. This is largely due to the fact that each case is somewhat unique - developed according to a particular need, the source and target 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 is usually chosen only during the specification of the transformation. Often also the semantic gap between the source and target languages of such transformations is larger than in traditional compilers.

The development of critical real-time embedded systems is guided by normative guidelines. Some of them are generic such as IEC 61508, while others are domain specific: DO-178 for aeronautics, ISO 26262 for automotive, EN 50128 for railway etc. They all require that system development is driven by precise specifications and that each system development step is verifiable and appropriately verified. However, the common proofreading of the detailed development artefacts and test-based V & V of the final products are usually not exhaustive. As systems get more complex, their coverage is less and less significant. Using modeling instead of informal specifications makes it possible to carry out early V & V and build *"correct by construction"* systems by doing automated model transformations. However, to provide confidence in the final product, it is highly desirable to also maintain traceability between the elements of the source and target models. Additionaly, the more critical the developed system, the more important is the clear separation between its *specification*, *implementation* and *verification* in terms of the development process, artifacts and individuals. In particular, independence is manda-

tory between the *implementation* and *verification* in order to reduce flaws both in the implementation and specification.

The current paper is based on one of the formally motivated studies related to the development of the open source qualifiable automated code generator GENEAUTO that transforms subsets of SIMULINK[2] / STATEFLOW[3] and SCICOS[4] modeling languages to C and ADA code. Previous work by one of the authors [10] focused on the use of a proof assistant for the development of the SIMULINK block sequencer in GENEAUTO. This experiment was technically successful, however it raised significant concerns regarding the predictability of development costs and transfer of the developed components to traditional software industry. To address these problems, this work focuses on common MDE techniques to allow industrial system and software engineers to write precise specifications and conduct verification at each step. More precisely, it relies on the *"Transformation Model"* approach similar to the one in [1] in order to provide both verifiability and traceability.

The approach we present is a lightweight syntactic/structural way for specifying the transformations in MDE style as model transformations and refine the details of the transformation relations by constraining them using the standard Object Constraint Language (OCL)[5]. An important part of our approach is the usage of explicit links between certain input-output elements to facilitate the specification and verification of the transformation. The work has been applied on some preprocessing steps of SIMULINK and SCICOS models in GENEAUTO.

The paper proceeds as follows. Section II discusses briefly the general transformation specification and verification process. Section III gives an overview of the approach presented in this paper. Section IV describes a case study. Sections V, VI and VII discuss some additional aspects of the approach. Section VIII positions the approach relative to existing works. Section IX opens up some extensions and summarizes.

[2]www.mathworks.com/products/simulink
[3]www.mathworks.com/products/stateflow
[4]www.scicos.org
[5]www.omg.org/spec/OCL/2.2

II. ABOUT TRANSFORMATION VERIFICATION

There are two general approaches to prove that the implementation of a transformation tool matches its specification: (a) verification is done once for all possible inputs to the tool or (b) verification is performed after each run of the tool. The first approach is more appealing as no verification activities are conducted when the tool is used. But, often the second one is simpler to implement. Many studies of the first approach have been conducted using theorem provers or proof assistants. The CompCert C compiler [12] is the most impressive result. But the cost of such a process is very high and not yet applicable in a typical industrial process.

The second approach called *Translation Validation* was proposed by A. Pnueli et al. in [15]. In this approach each transformation run is followed by a verification phase. The transformation phase should even provide hints about the way it built the target with respect to the source in order to ease the verification. However, the correctness of the verification should not depend on the hints. The main advantage of the approach is simpler specification of the properties of interest and a simpler verification tool. Hence, the approach has advantages, when the requirements for the transformation tool are complex and evolving, as is often the case for real compilers or domain specific program transformers. Also, the translation validation approach supports by definition a natural decoupling between the implementation and verification, which is well suited for the development of systems, where such independence is required. Finally, since the verifier is a simpler tool than in the first approach, it is simpler to verify the verifier itself.

III. TRANSFORMATION VALIDATION WITH TRANSFORMATION MODELS

A. Transformation Metamodel

In MDE, languages are specified with metamodels and each model must conform to its metamodel. This conformance is only structural. The core of MOF allows only to express simple structural properties, like associations between elements, containment, cardinality etc. OCL is an OMG standard that is designed for using with

UML and MOF and allows to complement metamodels with more complex constraints like structural invariants for classes and relations and pre-/postconditions for operations. When the model transformation is specified by a metamodel these constraints become correctness properties of the transformation. This is following the spirit of the *Design By Contract* paradigm.

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.

The specification of a model transformation contains three essential parts: definition of the source language, target language and transformation relation. In our approach the significant part of the transformation relation is modeled by explicit links between the elements of the source and target languages. These links correspond to the hints that the *Translation Validation* methodology expects from the transformation to simplify verification. Figure 1 shows our generic transformation metamodel. It has the three main parts stated above: references to the source and target models and bidirectional relation links between selected elements of the source and target models. The transformation tool must provide these links.

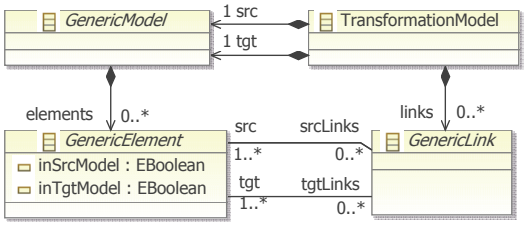

Fig. 1. Generic Transformation Metamodel

The input and output languages are given by their metamodels that are possibly complemented with OCL constraints. Our transformation metamodel assumes that the root elements in this model implement the abstract class *GenericModel* and all linked elements implement the abstract class *GenericElement*. This simple addition can be added automatically to any existing metamodel. To simplify the writing of the model transformation constraints, a derived attribute *inSrcModel* and its inverse *inTgtModel* have been added to the *GenericElement* class.

For the convenient specification of completness and correctness properties, backlinks from model elements to links are also provided. These backlinks can be built automatically during the verification phase.

The *GenericLink* metaclass makes it possible to build n-ary relations between different model element types. It should be noted that this is a generic scheme and there are different ways to implement it. To be able to write static constraints on different kinds of links, we define specific link classes for each relation kind, when instantiating the transformation metamodel for a particular model transformation. These links will be illustrated in the example in section IV-A. Different link classes relate different kinds of source and target elements with different arities. Furthermore, there can be distinguished roles among source and target links, as in the GotoFrom2SignalLink class. This allows to distinguish roles and arities of linked elements and make the link constraints more readable. The *src* and *tgt* relations in the GenericLink super-class can be implemented as derived relations.

A valid transformation model needs to comply with the transformation metamodel and a set of associated OCL constraints. Some of these constraints just specify the basic consistency of a transformation model e.g below are some rules for the correctness of the links (represented partially).

```
context GenericElement
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 : ...
```

The first constraints enforce that each model element in the source model (resp. target model) is linked only to sourceLinks (resp. targetLinks). For transformation-specific constraints, we use the following general pattern: (a) additional class invariants are specified for the source and target metamodel elements stating which links they must have (b) the transformation's correctness properties are specified as class invariants of the link classes. Note that the invariants from (a) belong

also to the transformation specification and do not affect the input-output metamodel definitions. Thus they should normally be stored separately from the metamodels.

These explicit links play a key role in this proposal. The links must be given together with the transformation instance to enable easy verification of the transformation correctness and also to provide the traceability data required by the certification authorities. Thus, these links must be 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 can be also called a gray-box approach: we require some information from the transformation performer, but not all the details of the implementation.

The number and kinds of links that must be specified is transformation specific and depends on the properties that one wants to verify. For completeness, one should require that each source model element (or all elements of some element classes) are related to some target model element and vice versa. Ensuring the consistency of the specification in the general sense and/or relation to the semantics of the transformation is out of the scope of this technology by itself and requires, for example, formalization in a theorem prover. However, this is an expensive step that is not always required or feasible, for instance, due to the lack of formalization of the source or target language. Often, careful specification of the transformation constraints can ensure the preservation and verifiability of the intended properties in a semantically sound way. For instance, by checking confluence of related transformations and validity of the expected local properties.

B. Transformation verification

In this configuration, the correctness of the transformation instance can be easily verified by checking that: a) the source model conforms to its metamodel and associated constraints (optional as it is a precondition for the transformation that is checked previously); b) the target model conforms to its metamodel and associated constraints; c) the transformation model structurally conforms to its metamodel; d) all the required translation links exist; and e) all links satisfy the respective constraints. The last four are postconditions for the transformation. These checks can be carried out using any standard metamodeling framework and an OCL checker.

The main difficulty in structural correspondence verification of a transformation 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], [14], [13]. 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 IV-A. Explicit transformation links, on the other hand, make it trivial to identify the related elements. Not all elements of source and target models need to be explicitly linked. It is only required that the explicit relation is given for elements, where the location of the transformed elements in the target model doesn't follow directly from the relative location of the source elements in the source model.

An important practical property of this approach is that the transformation specification does not need to be complete to be usable for verification. It can initially address only parts of the transformation and other parts can be specified gradually.

Verification of the transformation instance with respect to its specification is performed automatically using a MOF and OCL checker. Even stronger results can be potentially obtained, when the soundness and completeness of the transformation specification itself wrt. to some semantic criterion is analysed by formal methods. This other level of verification is harder and less common than verifying the implementation against specification. However, when the *Transformation Model* approach is followed, the transformation specification is already formal, making its deeper analysis possible.

IV. GeneAuto case study

GeneAuto is an open code generator project for translating high-level modeling languages to textual programming languages [18]. Currently, it supports subsets of Simulink, Stateflow and

SCICOS as input and C and ADA language as output. It is intended to be used and qualified for safety critical embedded systems. In that purpose its design follows a modular MDE approach allowing to independently verify different transformation phases.

There are about 50 to 60 transformation steps in the tool, depending on the configuration. After the initial importing step, all transformations are carried out as refinements and transformations of intermediate models. Most of these steps are rather simple, but some transformations are rather complex or change significantly the model structure. For practical purposes, only certain points in the transformation chain are observable as intermediate model files. The low-level tool requirements (transformation specification) are written with respect to these intermediate models, which are often a result of several successive transformations. Hence, verifying the structural correspondence between the input and output models is non-trivial. Explicit transformation links provided by the transformation tool can help overcome this, as shown by the example presented later in the paper and provide the traceability required by certification.

Currently, the specification for most of the elementary tools in GENEAUTO have been written 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]. Natural language requirements are of course incomplete, ambiguous and not directly verifiable.

A. *Preprocessing dataflow diagrams*

In the example we shall refine the transformation requirements for a component of GENEAUTO called Functional Model Pre-Processor (FMPre-Processor), which handles normalizing and refinement of dataflow diagrams. Figure 2 shows a section of the simplified Gene-Auto metamodel with the relevant elements.

In SIMULINK diagrams, signals can be split using Goto-From blocks and blocks can be grouped as *virtual* subsystems to avoid visual clutter. In the pre-processing step matching Goto-From blocks pairs are replaced by a signal; *virtual* subsystems boundaries and ports are removed; signals connected to these ports are connected directly to

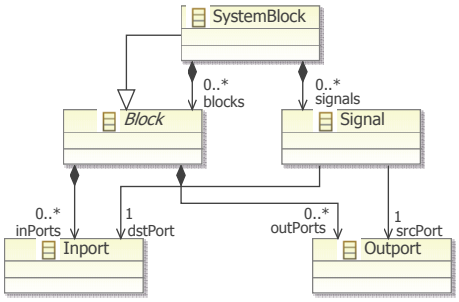

Fig. 2. A fragment of the simplified GASystemModel metamodel

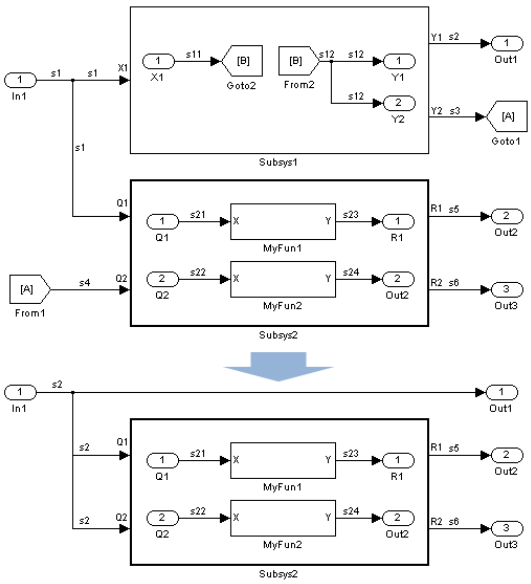

Fig. 3. Normalization of a dataflow diagram

inner blocks and redundant signal segments are removed. Figure 3 displays a diagram with chained Goto-From blocks, *virtual* subsystem Susbsys1 (thin border) and *non-virtual* subsystem Susbsys2 (fat border) before and after normalization.

Verifying a composition or transitive closure of such structure-altering transformations based on a source and target model only is complex and computationally expensive. For example, such analysis would have to build all signal paths in the source. Furthermore, it would not be possible to consider one transformation, such as Goto-From elimination, in isolation from e.g. subsystem

flattening or any other structure-altering transformation. This task gets much simpler, locally and compositionally verifiable, if the transformation outputs additionally the links binding the source and target elements.

B. Correctness of the transformation

In the case study we have formalized some of the natural language requirements in such a way that the specification can be directly used for transformation verification according to the scheme described in Section III. To specify the formal constraints for these normalizing transformations we have defined several link classes represented in figure 4. The first three concrete link classes relate objects of the same type with corresponding objects in the target model. The last one is more complex, relating a set of different elements (a configuration) in source model with elements (a configuration) in the target model. Link classes such as this one do not restrict the links to be simple binary relations and provide lot of flexibility for specifying the transformation while they also simplify verification and improve traceability.

The main OCL constraints that check the correctness of a Goto-From pair elimination are given below. Despite being formally precise, they should be rather readable and understandable to someone familiar with the nature of the given transformation. Notice the use of the *getTgtInDataPort* and *getTgtOutDataPort* operations. Such operations navigate from the source model to target model based on the elementary transformation links, which are essential for verifying the result of a composition of transformations, as they maintain the information between the original elements and their corresponding images in the target model. For instance, there are two ports corresponding to the input port of the block Y1 on Figure 3 in the normalized target model (ports are not shown). These ports are the input port of the block Out1 and input port Q2 of Subsys2.

```
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
```

These constraints state the facts that for *From* and *Goto* blocks, one of the related links must be a *GotoFrom2SignalLink* and if a *Block* is of *Goto* or *From* type then it is mandatory that the element is in the source model of the transformation.

```
context GotoFrom2SignalLink
inv gotoTagCheck :
  srcGotoBlock.GotoTag
    = srcFromBlock.GotoTag
inv tgtSigSameSrc :
  tgtSig->
    forAll(s | s.srcPort =
      tgtSig->first().srcPort)
inv gotoInSrcTransf :
  srcGotoSig.srcPort.getTgtOutDataPort
    = tgtSig->first().srcPort
inv fromOutDstTransf :
  srcFromSig.dstPort.
    getTgtInDataPort->asSet()
      = tgtSig.dstPort->asSet()
```

The first two constraints state that in the source model, the source port of the signal used as input of a *Goto* block, is transformed as the source port of the signal produced during the transformation. The last one verifies if the linked *Goto* and *From* blocks have corresponding *GotoTag* parameters (this parameter is used in order to find the paired elements - *Goto* or *From* block).

C. Verification infrastructure

Our experimental verification framework has been written in Java using standard components from the Eclipse Modeling Framework (EMF[6]) 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.

[6] http://www.eclipse.org/modeling/emf

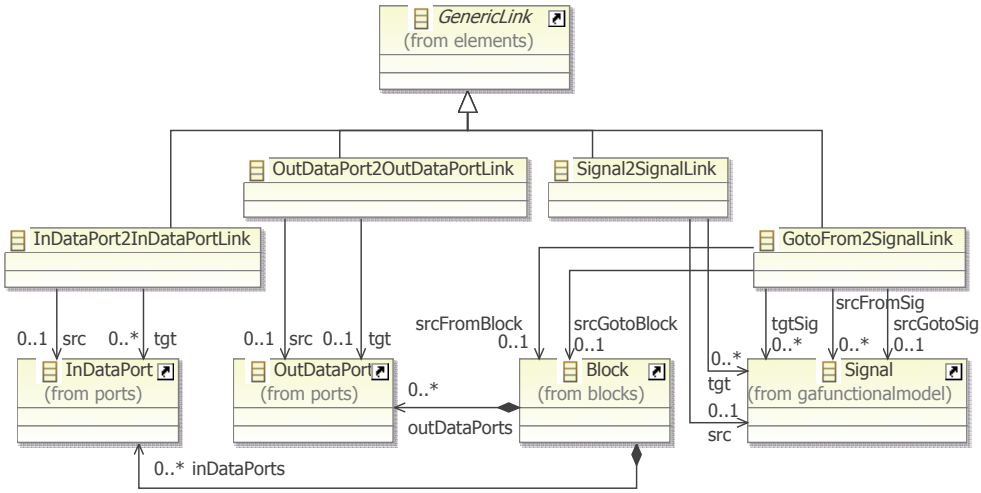

Fig. 4. Goto-From transformation links

Preferably, the formal specification should exist before the transformation is implemented. However, it is also possible to refine an informal specification and apply the presented approach at a later phase. In our case study the tool already existed. 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 and would have introduced minimal cost in the previous stages.

V. Relation to executable specification

It must be noted that technologically this approach has some commonalities with the *"executable specification"* approach, as e.g. in the QVT and ATL languages. In both approaches the source and target are specified by metamodels and the transformation is modeled as a relation between them. In our approach the transformation metamodel is used to verify that the transformation model is correct (transformation was correctly performed), while in the second approach it produces the output model. However, complex transformations are hard to represent in such way. 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 a tool like 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 I.

But on the other hand, since our approach presented above makes no assumptions about the transformation implementation technology, such formalization can be treated as a realization of the transformation specification and it can share the source and target metamodel definitions with the specification. The only additional requirement is that besides the output model it must also produce the required explicit transformation links.

VI. Analyzing the verification result

In real life and during the development process the implementation of the transformation or even the specification might be erroneous. If the *a posteriori* verification succeeds for a specific transformation run, the output is known to satisfy the specified properties. If the verification fails, and assuming that the specification is correct, one has the option to fix the transformation implementation, if possible, but also to fix the produced output: target model and/or links. For large and complex applications, fixing an error might be a

lengthy process and/or not under the control of the user of the tool, while it might be possible to correct the output instead. If the verification then succeeds, the user has a verified output with same guarantees as in the first case. Finally, a failure in the verification phase is likely to give messages that are understandable to the user. This is because the verification checks only structural properties of the source and target models and links. Hence, it is possible to immediately point the user to the violated OCL constraint and the concerned source and target model elements.

VII. Facilitating specification development

In refinement (or endogenous) transformations the source and target metamodels are the same. Such transformations have significant practical importance. For instance, in GeneAuto the majority of the 50 to 60 transformation steps are endogenous. Such transformations modify only a part of the model and one also needs to specify that the "other parts" do not change. This can be tedious for large metamodels. However, it can be partly automated. It is relatively easy to generate the specification (the link classes and related OCL constraints) of an identity transformation by using a general model-to-model transformation language like ATL or model-to-text language Acceleo. The specification can be then manually refined to take into account the required changes in the model.

VIII. 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). This section focuses on the use of model driven engineering technologies for the specification and verification of model transformation using the translation validation method. The main specific aspects are that: a) qualification with respect to certification standards is a key target; b) the global process including independent specification, implementation and verification activities must be handled; c) OMG standards are used for the specification and no other technological constraint should be enforced on the implementation; d) only structural properties are targeted. Semantic aspects will be handled in a separate phase done by different people than the one that implement the transformation (thus some specification V & V activities) using more appropriate technologies for the specification validation; and e) it is mandatory to handle industrial size models and transformations.

Many proposals rely on the use of formal methods and targets both structural and semantics issues. Regarding structural properties, declarative languages based on rewriting rules have been the subject of many proposals based on model checking technologies (see Varro et al. [16]) or rewriting technologies like confluence or termination checking (see Taentzer et al. [9], [11], [8]). However, these technologies do not scale well to industrial size models and transformations.

Bézivin et al. proposed in [1] to specify the model transformations using Transformation Models: the links between source and target languages are defined in metaclasses decorated with OCL constraints that express the correctness of the transformation. Their approach is methodological and is the main basis for our application on certified system development using model transformations. Braga et al. defined in [2] the notion of Transformation Contracts that are strongly related to Transformation Models and have applied it to security use cases.

Büttner et al. have also proposed recently in [3] 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. This verification approach is quite similar to the one from the previous paragraph. As the links are derived from the transformation implementation, the drawbacks previously stated still apply.

Narayanan et al. first proposed in [13] to use translation validation for verifying the correctness of model transformation. This work focused on the verification method and relied on the cross-links created as part of the transformation in the GReAT language much in the same manner as the traceability links from the Transformation Models. 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, ATL or Triple Graph Grammar [17] based tools such as Fujaba or Moflon. However, these links are mostly implementation links and usually much more links are produced than needed or the implementation must be very constrained to produce exactly the needed links. The work in [13] also focuses on semantic verification that is significantly more complex and cannot usually be handled by common software engineers. In order to handle partly these aspects when the structure of the source and target models are different, [14] proposes to rely on the link part of the Transformation Model: to specify the transformation as relations between the source and target metamodels. Then these links should be extracted automatically from the cross-links used for the implementation of the transformation in the GREAT language. There is still 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 is usually the case with declarative languages. Then it might get complicated to retrieve the specification links that are 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 or to only the implementation links? We propose to enforce the implementation to build exactly the right links that are precisely described in the specification. This introduces a cost on the implementation side, but also relieves the implementation team from the constraint of using a declarative transformation language. Moreover, as it is possible in the following proposal to reference the links in the transformation specification, complex and composed transformations can be specified and implemented quite flexibly. And in the end, these links are required by certification rules and thus the approach does not really introduces additional costs.

To avoid specifying explicitly the links, Cariou et al. have defined in [6] and [4], the specification of model transformations using pre and post conditions on the source and target metamodel expressed using OCL constraints. 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 anyway needed for the development of certified systems.

A completely different approach is that of the verified transformation tool. A well-known example here is the CompCert C compiler specified and verified in Coq by X. Leroy [12]. The last author with N. Izerrouken and X. Thirioux have experimented this approach in GENEAUTO for the block sequence elementary tool [10]. However, using such a methodology is still mostly in the domain of academic verification experts rather than domain engineers in the industry and hence it fits better for complex but more stable tasks, like a complier for a traditional language.

IX. CONCLUSIONS AND PERSPECTIVE

In this paper we present a pragmatic scheme for specifying and verifying model transformations based on the *Transformation Model* approach, which relies on standard MDE technology and is hence suitable for industrial application. Our scheme involves the explicit definition and maintenance of certain transformation links to allow convenient specification and efficient verification of the transformation. The methodology promotes early formalization of the transformation requirements and a natural independence between the specifier, developer and verifier, which is mandatory for highly critical applications.

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 semantics expert, who can assess independently the soundness and consistency of the specification using formal techniques, e.g. theorem proving. The last step is eased by the fact that the requirements have already been specified in a formal language (MOF with OCL). However, 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.

This proposal is similar to *Translation Validation* [15]. It has the obvious drawback to perform verification after each transformation run. But, it has also advantages when the transformation specification is complex and subject to changes, as in many realistic tools. First, a verifier is usually much simpler to implement than a full correctness proof. Then, as we propose to express the specification structurally using standard MDE techniques, the approach is applicable also in an industrial context by common engineers. In a certified/qualified context, it is also important to be able to assess the verification toolchain itself. This is eased by the fact that the approach makes use of a rather lightweight toolchain and standard components. An experimental verification framework implementing the approach has been added to the GENEAUTO toolset using Java and components from the EMF framework. We have formalized the specification of some of the transformations in the GENEAUTO code generator and verified the correctness of their implementation according to the presented scheme.

The developed tools have been integrated into the GENEAUTO testing verification framework. However, only a limited number of transformations have been currently specified in this style. The resources related to the case study can be found at `http://cs.ioc.ee/~toom/verification`. We plan to refine the approach and apply it on a wider scale and different set of transformations in the GENEAUTO continuation projects PROJECTP[7] and HI-MOCO[8]. It is of our special interest to study further the relations and the benefit such a scheme can bring to the certified software development process.

REFERENCES

[1] Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? Transformation models! In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 4199, pp. 440–453. Springer (2006)

[2] Braga, C., Menezes, R., Comicio, T., Santos, C., Landim, E.: Transformation contracts in practice. IET Software 6(1), 16–32 (2012)

[3] Büttner, F., Cabot, J., Gogolla, M.: On Validation of ATL Transformation Rules By Transformation Models. In: Proc. Workshop on Model-Driven Engineering, Verification, and Validation (MODEVVA'2011) (2011)

[4] 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)

[5] Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL contracts for the verification of model transformations. ECEASST 24 (2009)

[6] Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL contracts for the verification of model transformations. In: OCL workshop of MoDELS (oct 2009)

[7] Dave, M.A.: Compiler verification: a bibliography. ACM SIGSOFT Software Engineering Notes 28(6), 2 (2003)

[8] Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination criteria for model transformation. In: FASE. Lecture Notes in Computer Science, vol. 3442 (2005)

[9] Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: ICGT. Lecture Notes in Computer Science, vol. 2505 (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] de Lara, J., Taentzer, G.: Automated model transformation and its validation using atom 3 and agg. In: Diagrams. Lecture Notes in Computer Science, vol. 2980 (2004)

[12] Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)

[13] Narayanan, A., Karsai, G.: Towards verifying model transformations. Electr. Notes Theor. Comput. Sci. 211 (2008)

[14] Narayanan, A., Karsai, G.: Verifying model transformations by structural correspondence. ECEASST 10 (2008)

[15] 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)

[16] Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: A comparison of two approaches. In: ICGT. Lecture Notes in Computer Science, vol. 3256 (2004)

[17] Schürr, A., Klar, F.: 15 years of triple graph grammars. In: ICGT. Lecture Notes in Computer Science, vol. 5214 (2008)

[18] 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: ERTS[2]. Société des Ingénieurs de l'Automobile (2010)

[7] `http://www.open-do.org/projects/p/`

[8] `http://www.eurekanetwork.org/project/-/id/6037`