Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report*

Dominique Blouin[1], Alain Plantec[2], Pierre Dissaux[3], Frank Singhoff[2], and Jean-Philippe Diguet[1]

[1] Lab-STICC, Université de Bretagne-Sud, Centre de recherche, BP 92116 56321 Lorient CEDEX, France
{dominique.blouin,jean-philippe.diguet}@univ-ubs.fr
[2] Lab-STICC, Université de Bretagne-Occidentale, 20 av. Le Gorgeu, 29238 Brest CEDEX, France
{alain.plantec,singhoff}@univ-brest.fr
[3] Ellidiss Technologies, 24 Quai de la Douane, 29200 Brest, France
pierre.dissaux@ellidiss.com

Abstract. We report our experience of using Triple Graph Grammars (TGG) to synchronize models of the rich and complex Architecture Analysis and Design Language (AADL), an aerospace standard of the Society of Automotive Engineers. A synchronization layer has been developed between the OSATE (Open Source AADL Tool Environment) textual editor and the Adele graphical editor in order to improve their integration. Adele has been designed to support editing AADL models in a way that does not necessarily follow the structure of the language, but is adapted to the way designers think. For this reason, it operates on a different meta-model than OSATE. As a result, changes on the graphical model must be propagated automatically to the textual model to ensure consistency of the models. Since Adele does not cover the complete AADL language, this must be done without re-instantiation of the objects to avoid losing the information not represented in the graphical part. The TGG language implemented in the MoTE tool has been used to synchronize the tools. Our results provide a validation of the TGG approach for synchronizing models of large meta-models, but also show that model synchronization remains a challenging task, since several improvements of the TGG language and its tool were required to succeed.

Keywords: Model Transformation, Model Synchronization, TGG, MoTE, AADL.

1 Introduction

Model-Driven Development (MDD) often requires the use of many models to cover the various aspects of the system being developed. It is often the case that designers

* This work has been supported by the US Army Research, Development and Engineering Command (REDCOM).

D. Di Ruscio and D. Varró (Eds.): ICMT 2014, LNCS 8568, pp. 106–121, 2014.

need to describe the same system with different modeling languages to benefit from the assets of each language. For example, an embedded system model of the Architecture Analysis and Design Language (AADL) [1] may need to be translated into Simulink [2] for functional validation of the system through simulation. Users often want to be able to modify both the AADL and Simulink models and have their changes automatically propagated to maintain consistency of the models. In general, the information content of each model is not the same. One of the models may contain information that is not reflected in the other model because it is irrelevant for the purpose of the model. For example, an AADL model may include power consumption related properties, which are essential for power analysis of the system, but totally useless for functional validation with Simulink. Conversely, a simulation model may include details regarding the simulation, which are not needed on the AADL side. Hence, it is essential to be able to maintain the consistency of the models without the loss of the information that is not shared by both models. This is called model synchronization [19], which is a particular type of model transformation, as it operates on parts of the models at a finer level of granularity.

The need for model synchronization is becoming more and more important for the AADL community, and for model-based engineering in general, since more and more heterogeneous models are used together. For example, batch model transformations have been implemented between AADL and SysML [3], and between AADL and MARTE [4]. Another example is the Adele graphical editor [5], which implements the graphical syntax of AADL and operates on a meta-model of its own. It is the subject of our case study.

Although the need for model synchronization is widely spread, only transformation tools based on Triple Graph Grammars (TGG) can currently perform such type of transformation. Hence, the purpose of this paper is to report on our experiment on applying TGGs on a large and rich language such as the AADL. Our experiment validates the TGG approach for model synchronization, despite the many shortcomings that were identified during the work. Our contributions are:

- A synchronization layer between two AADL editors using different meta-models to represent AADL specifications.
- The new concept of generic TGG rules allowing to drastically reduce the number of rules needed for transforming models of large meta-models, thus improving scalability.
- A method to reuse existing model objects during synchronization thus avoiding information loss.
- Other minor improvements related to the expressivity of the TGG language.

This paper is structured as follows. Section 2 presents our model synchronization case study. Section 3 justifies the selection of the MoTE tool for our experiment. Next, an overview of the implemented solution is presented in Section 4. Section 5 briefly describes the Adele-AADL TGG and our contributions to the TGG language and MoTE tool. Section 6 discusses the tests that were performed on the synchronization layer, and suggests other potential improvements. Section 7 introduces the related work and finally, Section 8 concludes the paper.

2 Case Study

2.1 AADL

AADL is a rich component-based architecture description language that allows the capture of many aspects of an embedded system. The goal is perform model analysis in order to detect design errors early in the life cycle. AADL supports the specification of systems as an assembly of software and hardware components divided into categories. Software categories are thread, thread group, data, process and subprogram. Hardware categories are processor, virtual processor, memory, device, bus and virtual bus. Hardware and software components classifiers can be declared in libraries or hierarchically organized in systems for reuse. AADL components interact through features (interaction points) and connections, which together model data or control flows between components.

One advantage of AADL compared to languages such as UML is that it has both a textual and a graphical syntax. Users can therefore use the syntax they are more comfortable with, or the syntax that is best suited for whatever has to be edited in the model. Unfortunately, there has never been any usable graphical editor developed for the language, and it is a major drawback for the adoption of AADL. Graphical editors are complex, and every attempt to develop a graphical editor for AADL resulted in partially implemented tools that were barely usable.

2.2 AADL Editors

For instance, this was the case of the Adele graphical editor [5]. It stores AADL specifications using a meta-model of its own, which facilitates the edition of models by providing a choice of two edition modes. For example, Adele models can be edited in a top-down intuitive approach that does not follow the structure of the AADL language. Textual AADL specifications are generated from the Adele models for being processed by other tools.

Among these other tools, the Open Source AADL Tool Environment (OSATE) [6] is the main textual editor for AADL. It also provides model analysis facilities. Both Adele and OSATE can be deployed in the Eclipse environment for their simultaneous use. Unfortunately, Adele was never developed to the point where all constructs of the AADL language were managed. Constructs such as modes, flows, prototypes etc. have never been implemented, and sooner or later, Adele users were forced to use OSATE for editing these constructs through textual syntax. The result is that regenerating the textual files from the graphical model was then destroying the unhandled constructs. This was a serious issue because users were soon or later required to synchronize the models by hand. Three options could be envisaged to solve this problem:

- Implement the language constructs which are missing in Adele.
- Rebuild an editor that operates directly on the OSATE meta-model.
- Build a synchronization layer between OSATE and Adele.

Because model synchronization is an important need for the AADL community, the third solution was chosen. It would also provide a first case study of model synchronization with the AADL. In addition, it would allow preserving the more intuitive edition modes specific to Adele, and hardly supported by editors working on the OSATE meta-model directly.

2.3 The OSATE Meta-model

The OSATE meta-model contains 260 classes and is strongly typed. This is well illustrated by the way rules for components decomposition in terms of subcomponents are encoded in the meta-model. For example, an AADL system can contain subcomponents of many categories such as buses, data, devices, memories, processes, etc, as represented by the subcomponent containment references of Fig. 1. For each category (e.g.: *processor*), the containing class declares a specific reference to contain the subcomponents (*ownedProcessorSubcomponent*), and a specific subcomponent class for the category (*ProcessorSubcomponent*). AADL features (component interaction points), which are contained in component types are handled in a similar manner. As it will be explained in Section 5, this structure of the OSATE meta-model revealed several scalability issues, which were so important that they required implementing several improvements of the TGGs to validate their use.

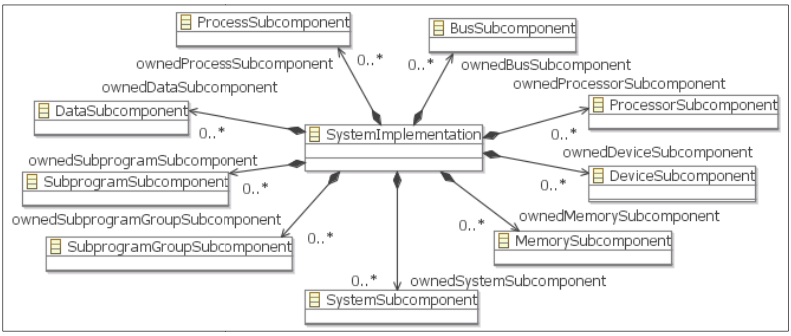

Fig. 1. A diagram of the AADL system implementation class showing distinct containment references and classes for every allowed subcomponent category

2.4 The Adele Meta-model

The Adele editor offers two modes to edit an AADL specification supported by two different types of diagrams. The declarative mode (package diagram) consists of declaring classifiers that can be instantiated later on to produce a system instance specification. Conversely, the instance mode (instance diagram) consists of creating a system instance specification, and to postpone the declaration of subcomponent classifiers to the time when they need to be reused for modeling other systems. In addition, a system diagram specification contains all subcomponents in a single tree.

To support these two edition modes, the Adele meta-model is used in two different ways. Package diagrams are edited according to the left part of Fig. 2, where the

hierarchy ends at the subcomponent level, and subcomponents of a subcomponent are declared in the classifier of the subcomponent. Instance diagrams are edited according to the right hand side of the figure, where all subcomponents are contained in a single tree whose root is the parent component implementation.

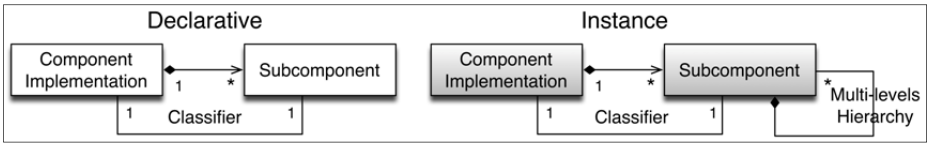

Fig. 2. The Adele meta-model and its uses for the declarative and instance edition modes

Compared to OSATE, the Adele meta-model is weakly typed and contains much fewer classes (48 compared to 260). For a given component category, the same class is used for representing component types, component implementations and subcomponents, where OSATE will declare three distinct classes (*ProcessorType*, *ProcessorImplementation* and *ProcessorSubcomponent*).

3 Model Synchronization Tool Selection

The first task of the project consisted of reviewing the available model synchronization tools. Our requirements were the following:

- The tool should be based on the Eclipse Modeling Framework (EMF), since both Adele and OSATE are implemented using this framework.
- The synchronization process should execute fast enough so that the user does not notice it.
- The objects should not be re-instantiated when changes are performed on the objects, with the instances being reused as much as possible and updated to restore consistency. This is to prevent information losses due to information not transformed by the TGG and contained in the destroyed object.

Several tools have been developed for model transformation. Well-known tools are ATL [9], Epsilon [10], Kermeta [11], Tom [12], and tools implementing the OMG Query View Transformation (QVT) language [13]. However, these tools only support classical one-way batch transformations. Model synchronization, where only part of a model is transformed (incremental transformation) is not supported. These tools are therefore not suitable to solve our problem, even though the relational part of QVT (QVT-R) is promising for model synchronization.

3.1 Approaches for Model Synchronization

As explained in [14], two main approaches exist for model synchronization. The first one considers that inconsistencies will naturally occur during design (e.g., the user modifies one specification without taking care of performing the corresponding modification on the other side), and means are provided to detect the inconsistencies

and automatically generate a set of actions to be applied on the models to restore consistency. This set of actions is called a repair plan, and inconsistencies are typically expressed by a set of constraints whose evaluation to true identifies inconsistent models.

The second approach uses coupled graph grammars. Consistency is characterized by membership of the models in the resulting graph language. Automatically generated operational transformations deal with maintaining the models consistent in case one model is changed. The most widely known language for this approach is the Triple-Graph Grammar [15]. Its power comes from the fact that the relation between the two models can be made operational so that models can be transformed / synchronized in either direction. While a graph grammar can be used for defining the dynamic evolution of a single model, a triple graph grammar allows to define the relation between two different kinds of models by defining and coupling three graph grammars (Fig. 4): one grammar for each type of model to be transformed, and a third grammar for a correspondence model whose purpose is to maintain traceability links between elements of the two models to be synchronized.

3.2 Tool Selection

Among the two approaches, only the TGG appeared to be mature enough. We could not find any tool implementing the first approach. On the opposite, TGGs have been around for more than 15 years, and several model synchronization experiments have been performed such as [16] and [17]. Our selection of the TGG tool has been strongly based on [19], which presents a survey of the three most widely known TGG tools still actively developed: (1) *The Model Transformation Engine* (MoTE) [7], (2) *The TGG Interpreter* [20] and (3) *The eMoflon* tool suite [21].

Both MoTE and eMoflon compile TGG rules into story diagrams. For MoTE, the story diagrams are interpreted to transform the models, while for eMoflon, Java code is generated from the story diagrams, which is then executed to perform the transformations. The TGG interpreter works differently as it directly interprets the TGG rules to perform the transformation. This has the advantage of allowing testing the transformation at development time, but has the drawback of reduced performance. Indeed, the survey indicated that that the TGG interpreter is the slowest, while MoTE and eMoflon have similar performances, with eMoflon being slightly faster than MoTE.

All three tools impose restrictions on the input TGGs for being able to prove correctness, and to increase performance in pattern matching in the case of MoTE. MoTE imposes the strongest restrictions on the TGGs. Both MoTE and eMoflon have proven completeness[1]. However, a serious drawback of eMoflon is that it did not support incremental transformations at the time of the survey, which prevented its use.

[1] Completeness means that every graph of a graph language $\mathcal{L}$(TGG) of a given TGG can be generated by the tool's forward/backward transformation from a graph of the translator's input domain.

While the survey did not strongly favor any of the three tools, MoTE appeared to be the best choice to synchronize Adele and OSATE. It supports incremental transformation, is fairly fast, has good formal properties and is completely based on EMF. Although a new version of MoTE (MoTE 2) improving performance and expressivity is currently under development, our work was performed with the version 1. This is because at the time we did this work, no TGG graphical editor was available for MoTE 2. At some point during the project, we considered porting our grammar to MoTE 2, but ran into several problems with the MoTE2 development tools and runtime execution. For this reason, we decided to complete the work with MoTE 1 first, and to postpone migration to MoTE 2 to a next phase of the project.

4 Overview of the Implemented Solution

This section introduces the architecture of the Adele-OSATE synchronization layer and its integration in the Eclipse-based modeling environment. We adopted an approach inspired from the work of [22], [23] and [24], where model synchronization is viewed as a specific task of Global Model Management (GMM). We have therefore developed a GMM language allowing formalization and interpretation of the various relations that can exist between models of a modeling environment. This includes the consistency relation between Adele and OSATE models, implemented as a synchronization relation using the MoTE TGG engine.

Fig. 3 presents the architecture of the Adele-OSATE synchronization relation and its deployment into the Eclipse workbench. A GMM controller listens for resource change events, which are sent by the resources manager of the Eclipse platform when users save the models through the editor. For a given changed resource, the GMM controller calls the GMM engine that processes the relations that concern the resource. These relations are declared in a GMM specification. The editor adapter layer is used to provide direct access to the internal resource of any opened editor of the resources to be synchronized, thus making the results of synchronization immediately visible in opened editors.

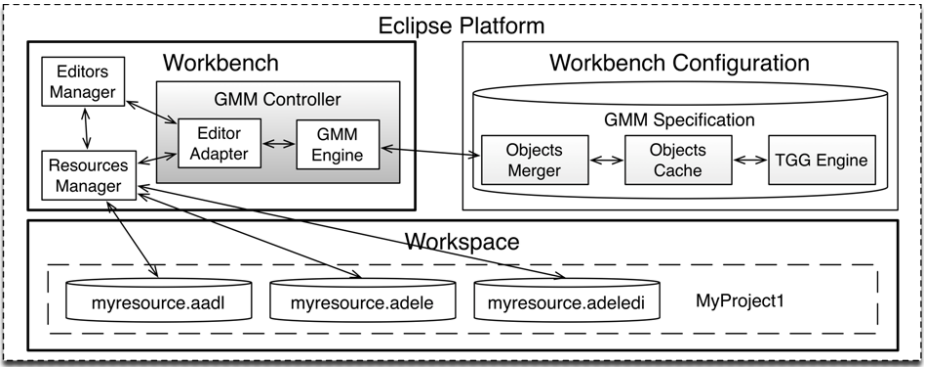

Fig. 3. The architecture of the Adele-OSATE synchronization layer, implemented as a synchronization relation of the Global Model Management language

Note that the MoTE synchronization relation makes use of a cache of the model objects, which are linked through the correspondence models for being synchronized. Changes made by any tool to the changed resource are merged into the cache, thus ensuring the objects traced by the correspondence models are not destroyed whatever the way the tool performed the changes[2]. Despite the fact that the merge operation increases complexity, it has the advantage but isolating the model objects to ensure that synchronization will work independently of the way the objects are modified. The merge layer is implemented using EMF Compare [25], which had to be tuned for merging models correctly as presented in Section 5.

5 The Adele-OSATE TGG

The TGG that was developed for synchronizing Adele and OSATE contains a total of 60 rules, as detailed in Table 1. The major portion of the AADL language has been covered, which makes our experience a relevant case study for applying TGGs to complex and rich languages. Most rules could be easily expressed, except for the connection rules, which required improvements of the TGG language and MoTE tool, and even modifications of the Adele meta-model. Therefore, this section focuses on these improvements, which unfortunately cannot all be presented due to the lack of space.

Table 1. Statistics of the Adele-OSATE

AADL Construct	# of Rules / Contexts	AADL Construct	# of Rules / Contexts
Package and public package section (axiom)	1	Subcomponents	11
Component Types	2	Connections	20
Component Type Features	10	Flows	Not Handled
Feature Group Types	4	Modes	Not Handled
Feature Group Type Features	10	Properties	Not Handled
Component Implementation	2	Prototypes	Not Handled
		Total	**60**

5.1 TGG Language Improvements

Generic TGG Rules

A first encountered problem relates to scalability of both the development and runtime tools, which could be fixed by introducing the concept of generic TGG rules. Indeed, while MoTE scales very well in transforming large models [19], we discovered limitations in handling complex languages like AADL. As illustrated in Fig. 4, a constraint on TGG rules is that the class of the created elements (green) must

[2] As a matter of fact, this need was initially discovered because the OSATE textual editor, which is based on the Xtext framework [8], systematically re-parses the AST as soon as any modification is made to the textual file.

be concrete (can be instantiated), and that the references to the created elements must be changeable. Without our improvement, the rule of Fig. 4 is not valid, since both the Adele component and AADL subcomponent classes are abstract, and the *ownedSubcomponent* reference is not changeable being derived. As introduced in section 2.3, the OSATE meta-model is strongly typed, and only the specific classes of each subcomponent category (Fig. 1) should have been used in TGG rules.

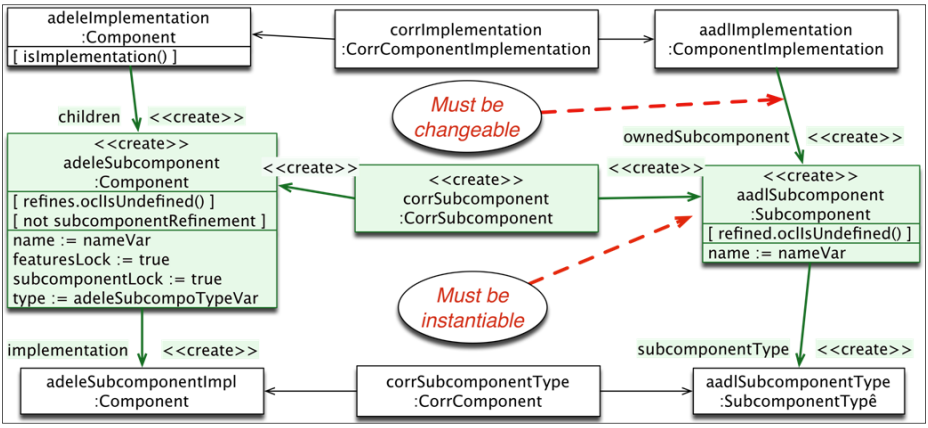

Fig. 4. The Adele-OSATE TGG rule for typed subcomponents

However, this quickly leads to an explosion of the number of required TGG rules. In order to meet the TGG "instantiability" constraints, a TGG rule would be required for each pair of parent component category and subcomponent category. For example, for the system parent category, 9 rules would be required to cover all allowed subcomponent categories. In addition, a subcomponent can be created in several contexts, which must all be covered by the rules. A subcomponent can be created with a type as illustrated in Fig. 4, or untyped as shown in Fig. 5, or typed as its parent, or with the subcomponent being inherited and refined to a more specialized type, according to the AADL subcomponent refinement mechanism. In total, the AADL language implies a number of 11 different creation contexts for a subcomponent of a given category. Hence, this implies that in order to cover only the system component implementation, 99 TGG rules are needed. However there are 14 component categories in AADL, and a simple calculation shows that more than 700 TGG rules would actually be required just for specifying the transformation of subcomponents!

Such a large number of rules cannot be handled by MoTE. First, the Story Diagram (SD) generator did not scale well with the number of TGG rules. It was observed that when a TGG reaches a number of roughly 300 rules, SD generation would require too much memory and would not complete on the computer used for this project, which had about 3.5 GB of RAM memory. Furthermore, the disk space required to store such a large number of SDs would make the release of the synchronization layer not manageable (about 3GB for 250 TGG rules). As a matter of fact, the space taken for a given synchronization SD increases with the total number of TGG rules, making the size of the total SDs not growing linearly with the number of TGG rules. SDs are

generated in a way that when a rule that produced a given set of objects is not matched anymore due to changes of the objects, a call is placed to all other rules of the entire grammar in order to discover a potentially matching rule. As a result, when the total number of rules increases in a TGG, the number of calls of the synchronization SDs increases and can potentially lead to performance issues.

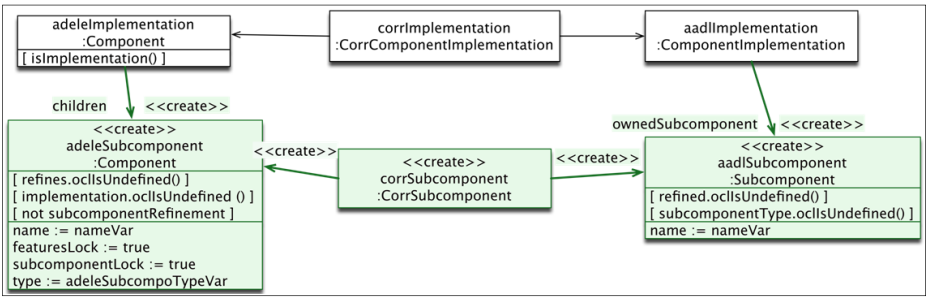

Fig. 5. The TGG rule for untyped subcomponents

To avoid these scalability issues, our first attempt was to optimize the SD generator to avoid consuming too much memory. But that turned out to be unnecessary after we implemented the concept of generic TGG rule allowing reducing the number of required rules for subcomponents from more than 700 to 11. This is shown in Fig. 4, which is the actual rule used in our Adele-OSATE TGG, and where the class *Subcomponent* of the created model element on the AADL side is abstract, and the containing reference (*ownedSubcomponent*) is not changeable in the AADL meta-model. The solution consisted of modifying the MoTE TGG language so that the TGG designer can provide an expression attached to model objects whose class is abstract, and to unchangeable model links. The SD interpreter then evaluates these expressions at runtime to determine which concrete class has to be instantiated, and which changeable reference has to be updated. In our specific case, the expression is a call action providing a static method of a transformation helper class called to determine the concrete class to be instantiated from the actual concrete class of the model element on the other side. A similar method is provided to determine the changeable reference from the types of the parent model element and the subcomponent.

Reuse of Objects to Avoid Information Loss

We also encountered problems with model objects being re-instantiated when synchronizing changes, despite the fact that MoTE had already been improved regarding this aspect. MoTE implements the algorithm presented in [26], which avoids re-instantiating the entire set of objects created in the sub-tree of the changed object.

However, changing a reference from an object to another model object caused re-instantiation of the object. For example, consider the rule of Fig. 4, which describes the creation of a subcomponent of a given type. When the subcomponent type is changed to null, the MoTE engine will detect that the rule that created the model element is not matched anymore due to the changes. In such case, it will try to match all other rules of the TGG. In our example, the rule of Fig. 5 (when the subcomponent is untyped) will

obviously be matched. MoTE will then repair the corresponding object by marking it as deleted, and by instantiating a new subcomponent and setting its properties according to the newly matched rule. This operation is performed with a dedicated SD named *repair structure*. While this is an improvement compared to the original algorithm, it is not sufficient for our use case. Elements such as AADL properties contained in subcomponents and declared on only one side of the TGG must be preserved. We therefore developed an original solution to avoid this re-instantiation (Fig. 6). It has the advantage of simplicity and limited overhead compared to other solutions such as that presented in [27].

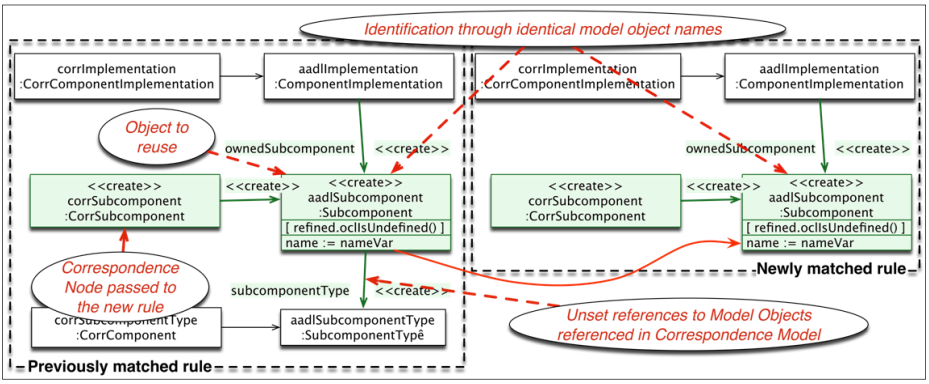

Fig. 6. The process of reusing an existing model object after changes

The objective is to be able to find the existing object that has changed so that it can be reused when applying the newly matched rule instead of instantiating a new object. Fortunately, when the repair structure SD of the newly matched rule is called, the correspondence node that refers to the changed object is always passed to the rule. We then simply need to identify the existing object to be reused from the set of objects referred by the correspondence node. For this, we can first discard all objects whose class is not the same as that of the object to be re-created. However, this criterion is not sufficient since there could be many objects with the same class created for a given correspondence node. To uniquely identify the changed object, we require an additional constraint to be verified by the TGG rules at the time they are defined. Comparing the rules of Fig. 4 and Fig. 5, we notice that their patterns are quite similar, and the only difference between them is that for the first one, there is a link to the type of the subcomponent, while for the second one, this link is removed and a constraint stating that the subcomponent type is null is added. If we require that the created subcomponent model objects have identical names for both rules (*adeleSubcomponent* and *aadlSubcomponent*), which is typically the case when good practices are used in TGG rules definition, we can then identify the existing object to be reused from the name of its model object in the newly matched rule. However, to make this possible, the way the correspondence model is represented has to be changed to associate with each created object the name of the model object of the creation rule. This is easily implemented in MoTE.

Now that the existing object is identified, two additional steps must be performed. First all references of the reused object to objects that are already mapped in the correspondence model must be removed. This is because the proper references will be set as if the object was newly created when the repair structure SD is applied. So we avoid setting the same object twice in case of multiple cardinality references. Conversely, objects that are not mapped in the correspondence model such as properties will not be reset by this process and will be preserved as desired.

This simple solution appears to work quite well for our Adele-OSATE TGG and ensures that whatever was contained by the subcomponent and not handled by the TGG is preserved. The only drawback is that it enforces using the same name in the created model objects of all rules for a given type of created model object. However, this can be ensured by adding constraints to the TGG meta-model.

Other Improvements

Other improvements were implemented but only briefly mentioned due to lack of space. We enhanced the MoTE global pattern matching constraints, by adding an applicability clause to the constraint. It states whether it is to be applied only during forward or reverse transformations. We also added the capability to specify an additional reference to a model object link to be used for matching purposes. In some cases, MoTE tries to match a pattern from the inverse direction of a model object link, starting from the target object to the source object. If the reference of the link has an opposite reference or is a containment reference, MoTE uses the opposite or the container reference to navigate to the source object. However, for some references of the OSATE meta-model, opposite references exist but are not explicitly specified in the meta-model. We therefore added a property on the TGG model object link class to specify this reference, and modified the SD interpreter for making use of it.

5.2 Tooling Improvements

Cross-Resource References Management

Another major issue with MoTE is that cross-resource references are not handled. For example, if a component refers to a component contained in a different resource, its reference property will not be handled during transformations. Such a shortcoming is more than enough to prevent a tool from being used. For languages like AADL, which provide packages declared in separate files to better organize a specification, this is a blocking limitation.

To overcome this problem, we modified the MoTE TGG engine so that it takes care of pre-building the correspondence model with correspondence nodes of external model elements. This fix slightly increases execution time, since a complete correspondence model has to be recursively created for each cross resource. However, each correspondence model is stored in the TGG engine cache so that the impact on performance is limited.

EMF Compare Improvements

As illustrated in Fig. 3, the integration of MoTE in our modeling environment is achieved through the use of a model object cache. This is needed to ensure that the model elements traced by the correspondence model are maintained. EMF Compare

1.3 has been used to merge the changes detected in the resources of the workspace into the resources of the cache.

Again, a few improvements were needed for being able to merge the models correctly. The first issue relates to the way EMF Compare merges changes in which cross resource references are involved. Our synchronization layer requires that if a reference to an external element is set, then the external model element should be contained in a resource of the cache on the cache side, because it may already be referred by a correspondence model. However, the default behavior of EMF Compare is to set the same referenced object in the target object as that of the source object, and EMF Compare had to be adapted to take this into account.

Another problem is related to the order in which the merge operations are performed. To avoid unnecessary re-instantiation of objects, the delete operations, which are received as model change events by the TGG engine, should be added at the end of the transformation queue. In this way, the references to the object to be deleted can be moved to other objects before the object is deleted during synchronization. Hence, this implied modifying the order of the merge operations in EMF Compare to ensure that deletion operations are performed at the end. Other minor merging issues were also identified and corrected but cannot be presented due to the lack of space.

Other Improvements

Other improvements to MoTE were required as well, such as the implementation of post-creation actions, which were declared in the TGG language but not implemented in the TGG compiler. The same is true for position constraints used to define whether an element is to be added at the first or the last position in a list. This feature was handled in the SD interpreter but not in the TGG language. We also introduced the *between* constraint used when the element should neither be the first and nor the last element, provided that there are at least two elements in the list. We also had a few issues with the MoTE change listener that receives model changes events to be added as modifications to the TGG engine transformation queue for synchronization.

6 Discussion

6.1 Implemented Synchronization Layer

The concrete result of this work is a synchronization layer between Adele and OSATE, solving the problem of integrating tools that are essential for the growing AADL community. Experience gained during this project supports the ongoing work of integrating other languages and tools with the AADL such as VHDL.

Automated tests were developed for the synchronization layer for testing bidirectional batch transformations and synchronization transformations, and for consistency checks performed by creating and analyzing correspondence models. In addition, we have tested our synchronization layer with several realistic and complex AADL models such as electronic hardware systems using all constructs of the AADL language handled by Adele. Such systems require up to 7 levels of recursive component extensions declared in different files, which was used to validate our fix of the MoTE for the cross-resource references problem.

6.2 Suggestions for Further Improvements

Based on our experiment, we found that current TGG approaches require further improvements for being suitable for industrial use. These would increase the usability of TGGs and ease the development of model synchronization layers. For example, it would be useful to have a mechanism to allow reusing one side of an existing TGG (for instance the AADL side) and complete the other side according to the new language to be synchronized with AADL (e.g., VHDL). Another improvement could be to provide a "soft" reference mechanism for model links of graph patterns, instead of requiring the reference to be declared in the class of the source model object. In our experiment, this would have avoided the need to add references to the Adele meta-model, which are used only for TGG rule matching purposes. This is even more important for the cases where the modeling languages cannot be modified (e.g.: the AADL). In addition, other improvements published in the literature would really need to be implemented in MoTE. These are described in the related work section.

It was also found that the use of TGGs could be made much easier if better documentation was provided. In the actual state, the transformation designer needs to understand the generated SDs to be able to define correct TGGs, and several additional TGG validation rules would need to be enforced. For example, when defining constraints, beginners have no clue which model objects can be used in constraints. The bound model objects depend on the specific type of transformation (mapping, batch or synchronization), and several errors occur at runtime due to unbound model objects being referenced in constraint expressions.

7 Related Work

To our knowledge, no experience has been made to assess the usability of TGGs for synchronizing models of complex and rich languages such as the AADL, with a real need to integrate tools used by an active community. However, a few similar works can be compared to ours.

In [19], a set of benchmarks has been performed for large models to compare three TGG tools (MoTE, eMoflon and the TGG Interpreter). However, the meta-models are extremely simple. In [16], synchronization has been implemented with MoTE between SysML and AUTOSAR, but only subsets of the languages were covered. A work closer to ours is presented in [17], where the synchronization of Modal Sequence Diagrams (MSDs) with networks of Timed Game Automata (TGA) using the TGG Interpreter is presented. Like for us, their experiment lead to the development of several improvements such as:

- The integration of OCL in TGGs, which is already implemented in MoTE.
- TGG rule generalization / inheritance, which is also introduced in [18] for eMoflon, and would be worth implementing in MoTE.
- Reusable patterns, which allow declaring in a single TGG rule several contexts of creation of a given graph of model elements. This would have helped in MoTE by reducing the number of required rules for handling the numerous creation contexts (e.g., 11 for subcomponents).

- Global constraints, which are already implemented in MoTE, to which we added an applicability clause for specific transformation directions (forward / reverse).

Furthermore, in [27], a new algorithm is presented and implemented in the TGG Interpreter to allow further reuse of model elements during synchronization. It avoids the loss of the information not handled by the TGG rules. We provided a different solution to this problem, which appears to be simpler but requires additional constraints to be met by a set of TGG rules describing the contexts of creation of a given model object class.

8 Conclusion and Perspectives

In this paper, we reported our experience in synchronizing models of two different meta-models for the complex feature-rich AADL language. Our experiment shows that applying state of the art model synchronization techniques remains challenging, despite the good quality of the MoTE tool that was used. This case study allowed the development of several improvements of TGGs to account for the size of the AADL meta-model and its properties. However, the fact that we succeeded in synchronizing the tools validates the TGG approach and opens interesting perspectives.

A future work will consist of porting our improvements and our Adele-OSATE TGG to MoTE 2, in order to benefit from the MoTE 2 improvements. In addition, we are currently working on improving other aspects of TGGs and MoTE through the development of an endogenous refinement transformation for AADL, and an AADL-VHDL transformation. We also plan to write a cookbook to help developers get acquainted with TGG development.

References

1. SAE International, Architecture Analylsis and Design Language (AADL), http://standards.sae.org/as5506b/
2. MathWorks, MathLab Simulink, http://www.mathworks.fr/products/simulink/
3. OMG, Systems Modeling Language (SysML), http://www.omgsysml.org/
4. OMG, Modeling and Analysis of Real-Time Embedded Systems (MARTE), http://www.omgmarte.org/
5. The Adele Graphical Editor for AADL, https://wiki.sei.cmu.edu/aadl/index.php/Adele/
6. Open Source AADL Tool Environment (OSATE), http://www.aadl.info/aadl/currentsite/tool/osate-down.html
7. The Model Transformation Engine (MoTE), http://www.mdelab.de/mote/
8. The Xtext Framework, http://www.eclipse.org/Xtext/
9. The Atlas Transformation Language (ATL), http://www.eclipse.org/atl/
10. The Epsilon Project, http://www.eclipse.org/epsilon/
11. The Kermeta Project, http://www.kermeta.org/
12. The Tom Project, http://tom.loria.fr
13. OMG, Query View Transformation (QVT), http://www.omg.org/spec/QVT/

14. Boronat, A., Meseguer, J.: Automated Model Synchronization: A Case Study on UML with Maude. Proc. of the ECEASST (41) (2011)
15. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
16. Giese, H., Hildebrandt, S., Neumann, S.: Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 555–579. Springer, Heidelberg (2010)
17. Greenyer, J., Rieke, J.: Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 222–237. Springer, Heidelberg (2012)
18. Klar, F., Königs, A., Schürr, A.: Model Transformation in the Large. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE 2007), pp. 285–294 (2007)
19. Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schafer, W., Lauder, M., Anjorin, A., Schürr, A.: A Survey of Triple Graph Grammar Tools. In: Proc. of the 2nd International Workshop on Bidirectional Transformations (2013)
20. TGG-Interpreter, `http://www.cs.upb.de/index.php?id=tgg-interpreter/`
21. eMoflon, `http://www.emoflon.org/`
22. Hebig, R., Seibel, A., Giese, H.: On the Unification of Megamodels. In: Proc. of the 4th International Workshop on Multi-Paradigm Modeling (MPM 2010). ECEASST, vol. 42 (2011)
23. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing in Model Management. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 197–212. Springer, Heidelberg (2009)
24. Seibel, A., Neumann, S., Giese, H.: Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. Softw. Syst. Model 9(4), 493–528 (2010)
25. EMF Compare, `http://www.eclipse.org/emf/compare/`
26. Giese, H., Hildebrandt, S.: Efficient Model Synchronization of Large-Scale Models, Tech. Rep. 28, Hasso Plattner Institute at the University of Potsdam (2009)
27. Greenyer, J., Pook, S., Rieke, J.: Preventing information loss in incremental model synchronization by reusing elements. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 144–159. Springer, Heidelberg (2011)
28. Giese, H., Hildebrandt, S., Seibel, A.: Improved Flexibility and Scalability by Interpreting Story Diagrams. ECEASST (18) (2009)