Electronic Communications of the EASST
Volume 16 (2009)

Proceedings of the Doctoral Symposium at the International Conference on Graph Transformation (ICGT 2008)

Using a Triple Graph Grammar for State Machine Implementations

Michael Striewe and Michael Goedicke

15 pages

Guest Editors: Andrea Corradini, Emilio Tuosto
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Using a Triple Graph Grammar for State Machine Implementations

Michael Striewe[1] **and Michael Goedicke**[1]

[1]Universität Duisburg-Essen
{michael.striewe, michael.goedicke}@s3.uni-due.de

Abstract: Typical techniques of model-driven development use graph transformations to manipulate models and use generators to produce source code. In this contribution we suggest to use graph transformations instead of generators in order to get a closer connection between model and code. We define a Triple Graph Grammar for the mapping from a modeling tool data format to source code and derive a sample set of transformation rules from this. Thereby both truly simultaneous manipulation of model and code is enabled as well as virtually simultaneous manipulation by direct propagation of changes from code to model and back again.

Keywords: Triple Graph Grammar, Model translation, Finite State Machine

1 Introduction

A convenient way of modeling state- and process-oriented system behavior is the use of finite state machines. They can be comprehensively specified, simulated and validated at design time to get a formally founded skeleton for a software application. They integrate well with other approaches, because each transition may invoke either generated code from other models or arbitrary source code, possibly from legacy systems.

A need to preserve the semantics of the state machine model in the code emerges from the two possible types of errors that can occur: either the model itself is erroneous, e.g. leading to a deadlock, or the run time behaviour of the system does not meet the expected specified behaviour. In the first case, it is sufficient to check the model against the original specification. In the second case, there may be an erroneous model, an erroneous manual implementation or errors in generated source code. Hence, it is necessary to check the implementation as the final result after several steps against the original specification. This can be done be extracting a model from the implementation and using the same model checking techniques as in the first case, provided the extraction does not change any semantics. Viewed from an abstract level, implementation and model are two views on the same system, that have to be transformed simultaneously to keep the connection between them. To maintain the resulting software systems, it is vitally important to preserve as much of the semantic information of the models as possible in the source code to be able to track back changes and errors [BLW05, HT06].

A direct implementation of a state machine in object-oriented source code can realize states as classes, transitions as methods inside these classes and variables as auxiliary methods that invoke arbitrary application methods to retrieve the current variable values. Guards and updates for transitions can be realized as methods of boolean type, evaluating expressions and indicating whether a condition is fulfilled or an expected update has taken place, respectively [BSG08, BSG09]. Thus the complete semantics of a state machine can be embedded into source code.

In contrast to these considerations and possibilities, current techniques of model-driven development use several unidirectional steps of transforming and refining models, starting from an abstract model and resulting in platform specific source code. Semantics are not kept this way and hence there is no way back to re-generate the model from existing source code. Additionally, large systems may be composed of several interacting subsystems derived from different models. Changes in one of these models should only be allowed to change existing source code but not to override it, which could destroy the implementation of other models.

Following the ideas of [Sch94, Kö05], this problem leads to the idea of a Triple Graph Grammar (TGG). We can represent the view used for verification using the syntax graph G_V of some modeling tool data format. Similarly, we can represent the implementation using the syntax graph G_C of the chosen programming language. An unidirectional transformation from G_V to G_C is the classical model-driven development approach. A bidirectional mapping between both graphs via a correspondence graph G_M constitutes a TGG. Thus, G_M covers the idea and semantics of a concrete state machine, but no explicit representation in terms of a certain data format or programming language. It allows (1) to create G_C from a given G_V in order to realize code generation from models, (2) to create G_V from G_C in order to extract models from source code and (3) to manipulate G_V and G_C simultaneously in order to allow model-based editing of source code.

The expected benefit is to tackle all problems named above. Simultaneous editing allows to change the initial models in later design phases without destroying unrelated existing source code by newly generated code. Extracting models from source code allows to check the implementation against the original specification, provided the TGG itself preserves the semantics of the model. Finally, code generation is still possible with TGGs, so the new benefits do not come on the cost of loosing older ones.

This contribution is structured as follows: Section 2 summarizes related work in the context of TGGs and model-driven code generation. Section 3 explains the formal base of our approach in terms of type graphs and mappings and gives an example. Section 4 describes the actual TGG rules. Section 5 depicts a tool implementation based on our approach. Section 6 evaluates the approach and section 7 concludes our work.

2 Related Work

The approach presented in this paper is related to different research subjects. At the formal base it uses Triple Graph Grammars that are also studied in other application domains. In [JKS06] Triple Graph Grammars are used to specify views on models to avoid the duplication of data. Another approach using Triple Graph Grammars in view management to propagate changes to derived views is shown in [GL06]. This is partially similar to our approach, since we can propagate changes from model representations to implementations and the other way round. In [JZ99] Triple Graph Grammars are used for transformations between a conceptual data model and legacy data models for databases. This is related to our approach on a higher level, since model representations are more abstract than the implementations that are embedded into legacy systems. The synchronization of models is also addressed in [GW06] with special respect to incremental changes and their performance aspects.

Model transformation and code generation with explicit usage of syntax graphs or mapped models can also be done without the use of TGGs. Model refactoring for the Eclipse Modeling Framework (EMF), that can be used for code generation, is done by graph transformations in [BEK+06], including syntax graphs of source code as special model in [Tae08]. Different to our approach, this work does not use model semantics embedded in the source code. The system presented in [WS05] maps arbitrary UML models to Java source code, but traverses an UML tree instead of using a graph transformation. In general, unidirectional model transformation and code generation is well studied, but embedding semantical information explicitly into source code and extracting it later on is not a widely discussed approach.

However, simultaneous manipulation and model extraction is a crucial point for verification and re-engineering and hence also addressed by other techniques than TGGs. Model round-trip engineering [AC06, SK04] and queries for model-specific code patterns [ABC07] do partially the same as our approach, but cannot cover both propagation of changes and truly simultaneous manipulations as graph grammars can do.

3 Defining the approach

In this section, we describe the formal parts our approach is based on in terms of type graphs and mappings. To illustrate things, we give an example that will be used throughout the rest of this paper.

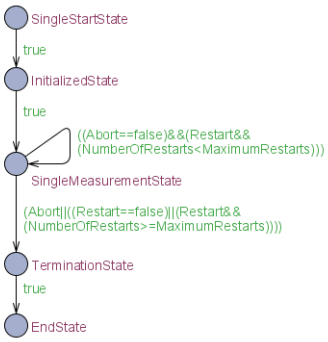
3.1 Graphs and Type Graphs

The Triple Graph Grammar used in this paper is based on syntax graphs both for modeling tool data format and programming language. The approach can be used with any language that can be parsed to a syntax graph. Thus our approach can also be used to translate between different data formats or programming languages. As an example, we use the simple XML-based data format from the real-time model checker UPPAAL [LPY97] for G_V throughout the rest of this paper. As a programming language, we use Java and its syntax graph for G_C.

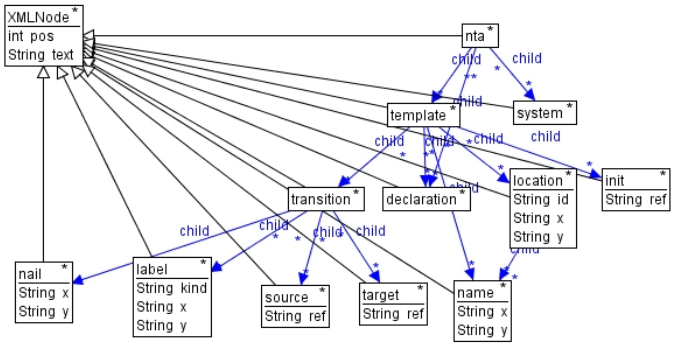
3.1.1 UPPAAL Syntax

The graph G_V used to represent the model is typed over the syntax elements for the modeling language. In case of a simple XML-based language, the necessary type graph can be derived from the related data type definition (DTD) of the XML format. The type graph for UPPAAL derived manually this way is shown in figure 1(b). To illustrate its usage, figure 1(a) shows a finite state machine modeled in UPPAAL and figure 1(c) the graph of its representation in XML.

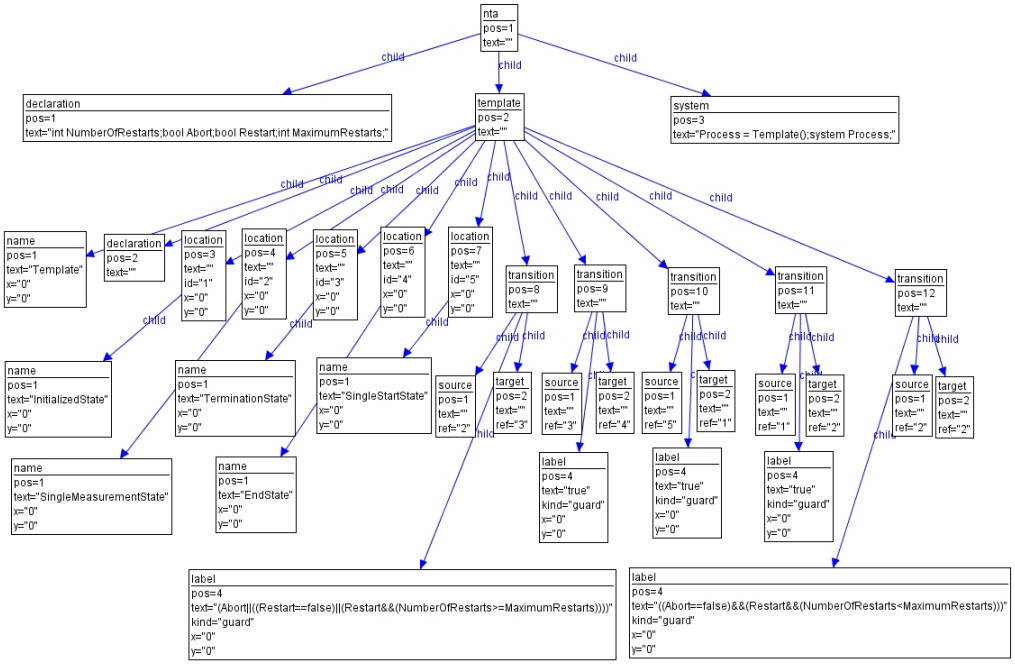
The type graph appears to be very simple, since it covers parts of the state machine in a rather abstract way. For each state, there is a node named “location” with a child for its name. For each transition there is a node named “transition” with children for its source, target, guard and update. Guards and updates are represented as nodes of type “label”, where the inherited attribute “text” holds an expression as string and another attribute “kind” indicates the usage as guard or update. This will require string parsing mechanisms during graph transformation to split up these strings



(a) A state machine with five states and five transitions modeled in UPPAAL.



(b) Type graph for the XML structure of the UPPAAL data format.



(c) Graph representing the XML structure for the state machine shown in (a) based in the type graph shown in (b). For each state, there is a node named “location” with a child for its name. For each transition there is a node named “transition” with children for its source, target and guard. Each location has an ID by which it is referred in sources and targets.

Figure 1: Example taken from an existing implementation of our approach, showing visual model representation in the UPPAAL editor (a) and the underlying XML syntax of the data format (c) based on type graph (b).

into more detailed syntax graphs for expressions. Since the result of these parsing operations could be expressed by an extended set of types in the DTD, this is no general limitation.

The node "nail" is only used for layout information as well as the attributes "x" and "y" in some of the nodes. They are without relevance for the semantics of the model. Constructing the transformation rules, it has to be kept in mind that attributes or graph nodes of types like these are not mapped to other graphs and hence are not translated into the other language. However, a concrete syntax tree of the given language might not be complete without these nodes and hence transformations must be able to include reasonable default nodes without state machine semantics if necessary. In contrast, if the grammar is used to manipulate an existing syntax graph, these default nodes may not override existing ones.

3.1.2 Java Syntax

The graph G_C representing the source code is typed over the abstract syntax elements for the programming language. This type graph can be derived from the according language specification. Because of the large size of the resulting type graph in the case of Java, we do not show a sample figure here. As usual for programming languages, syntax graphs for Java do not contain any superfluous layout information and even no source code comments. Thus, any semantical information that should be extracted from the source code has to be represented by real program statements and expressions. Figure 3 gives an example of the used syntax graph, representing the source code shown in figure 2. Details about the chosen Java constructs are out of the scope of this paper and can be found in earlier publications [BSG08, BSG09].

Although these graphs are already complex, it can be beneficial to extend them by additional elements for practical reasons. Consider a syntax graph having a node of type "SimpleType" to denote literals referring to a type declaration. The according declaration might be represented by a node of type "TypeDeclaration" somewhere in the syntax graph. Although both elements are semantically related there will be no edge between these two nodes, because they are syntactically independent. Adding a new edge type (e.g. named "access") to the type graph allows to connect the nodes. Some edges of this type are visible in figure 3. Semantic information is available for graph pattern matching by this means and hence mapping of structures and application of rules becomes easier.

3.1.3 Correspondence Graph

As depicted in the introduction, we would like to constitute a TGG by mapping G_V to G_C via a correspondence graph G_M. This graph has to contain node types for each semantically relevant element of a state machine, independent of any concrete representation in a data format oder programming language. Although we stated above that G_M has to be understood as an abstract view on the semantics of the state machine without a concrete representation, the type graph for G_M defines a language for representing this idea. This language forms the correspondence graph used in the TGG and provides the formal backbone of the TGG this way. The type graph is shown in figure 4. Obviously it has to contain nodes for states and transitions. Guards are understood as expressions, connected with a transition via an edge named "PRECONDITION". The same applies for updates, which can be either deterministic or non-deterministic. For deterministic

```
public class SingleMeasurementState implements IState
{
  @Transition(target = SingleMeasurementState.class,
              contract = RestartMeasurementContract.class)
  public void restart(MeasurementModule actor)
                  throws MeasurementAbortedException
  {
    actor.increaseNumberOfRestarts();
    actor.initClients();
    actor.doMeasure("Restarted measurement");
  }

  @Transition(target = TerminationState.class,
              contract = TerminateMeasurementContract.class)
  public void terminateMeasurement(MeasurementModule actor)
  {
    actor.terminateMeasurement();
  }
}
```

Figure 2: Class `SingleMeasurementState` with some outgoing transitions. This class implements the third state from the state machine shown in figure 1(a). I.e. it is marked to be a state by the `implements IState` fragment. It realizes two transitions, recognizable by the Java annotation `@Transition`, naming the target of each transition as well as separate classes where guards and updates are defined.

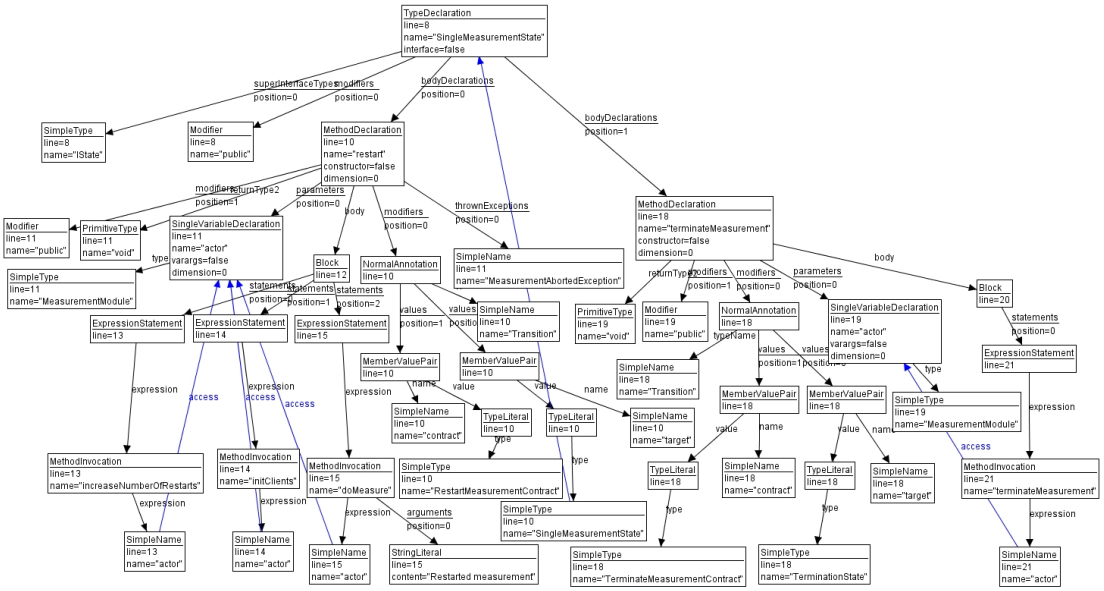


Figure 3: Syntax graph representing the source code shown in figure 2.

updates we know a fixed value that is assigned to a variable when a transition is fired. For non-deterministic updates, we only know the range a value is chosen from. Both ways of updates can be represented in G_M by the nodes "VALUEUPDATE" or "RANGEUPDATE" respectively. In general, the type graph of G_M captures expressions in a more detailed way than the one for UPPAAL (see figure 1(b)), offering different types for operators and literals as well as edge types with attributes for sorting operands. However, it does not allow as much combinations of expressions and operators as a Java type graph would allow.

We mentioned above the existence of nodes in syntax graphs G_V or G_C, that are not relevant for the state machine semantics and that are hence not mapped to G_M. Similar applies to elements of the state machine definition, that might not be supported in a certain modeling tool or programming language. These elements are present in the type graph for G_M, but it is possible that there will be no mapping for them into G_V or G_C. Nevertheless they cannot be neglected in G_M since they may be relevant when translating the state machine into a different language. E.g. a modeling tool may not support non-deterministic updates and thus there will be no mapping from the node "RANGEUPDATE" into G_M.

3.2 Mappings

In order to realize the desired transformations, two graph mappings have to be defined: one between G_M and G_V and one between G_M and G_C. The union of both mappings defines the bidirectional mapping between G_C and G_V. Figure 5 shows this mapping covering the correspondence for states and transitions by example. Elements denoting expressions for guards and updates are not presented in this figure.

In general, there will always be a set of nodes in a syntax graph, mapped to a single node in the correspondence graph, that is in turn mapped to a set of nodes in the other syntax graph. The single node in the correspondence graph has to collect all necessary attributes that have to be translated. In the sample figure this is shown by the attribute "name" in "STATE" nodes of the correspondence graph, that maps the values of the attribute "name" in node "TypeDeclaration" and "text" in node "name" as child of "location" in order to ensure properly named states.

Guards and updates would be represented by additional nodes according to the type graph shown in 4, connected to a "TRANSITION" node by edges of type "PRECONDITION" or "UPDATE". These edges point to nodes covering the syntactical structure of the expression. This avoids the need for string parsing mechanisms when translating expressions from one language to another. In the case of Java, these structures can directly be mapped to Java syntax. In case of UPPAAL, they are mapped to a node of type "label", which contains the expression in string form as an attribute.

Figure 5 also points out the relevance of additional edges in the Java syntax graph introduced in section 3.1.2. Using the edge of type "access", the reference from a transition to its target can be found in the Java syntax graph by graph pattern matching without attribute comparison. See the blue edge of type "access" on the left side of the figure, leading from a Java type name to its declaration and the according red edge in the correspondence graph, leading from a transition to its target.

Although graph pattern matching is sufficient to identify elements to be mapped, attribute comparison might be needed to distinguish between these constructs and similar structures that

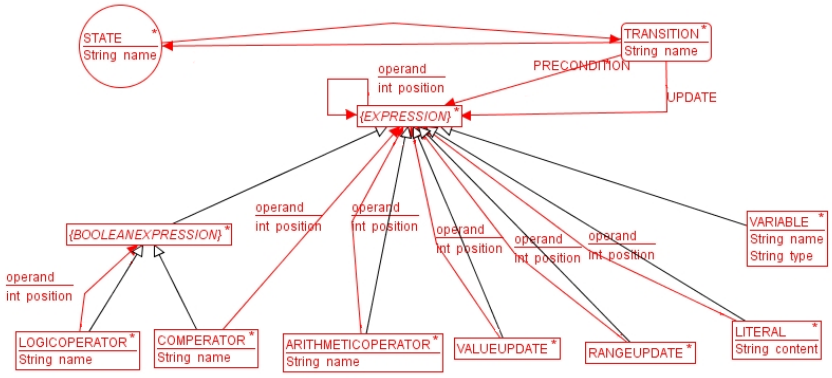


Figure 4: Type graph for the correspondence graph.

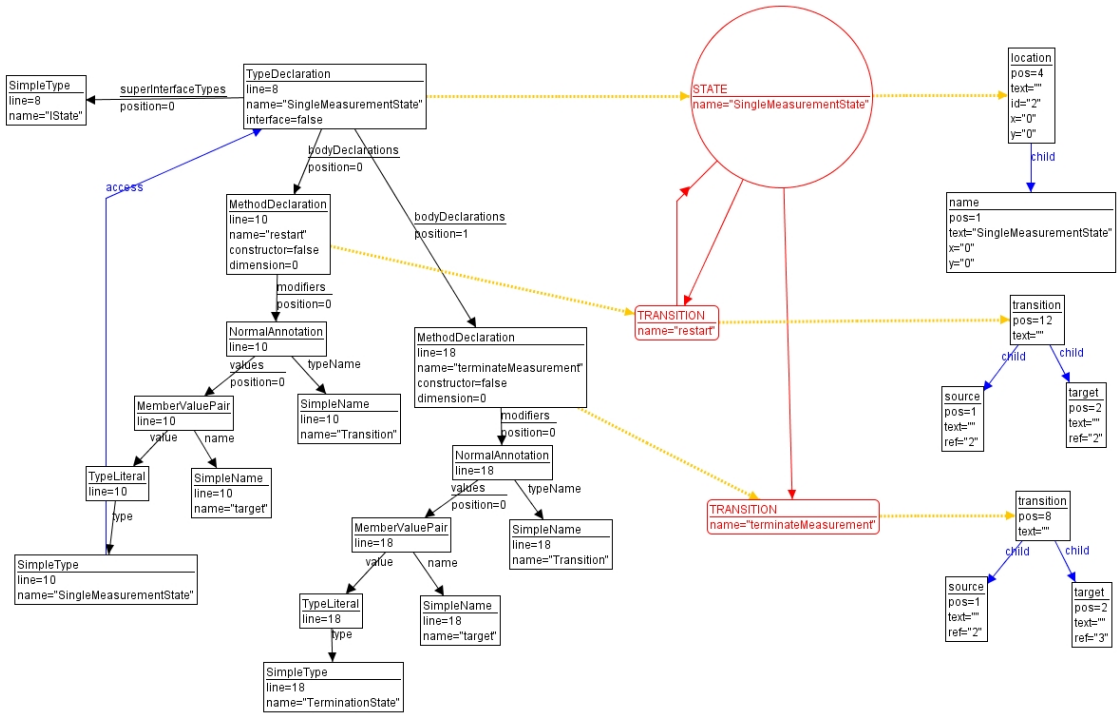


Figure 5: Mapping for the third state from the example state machine and its transitions from programming language (left) to XML format (right) via the correspondence graph (center). Directly mapped elements are connected by dashed yellow lines. All other elements are necessary to complete the respective syntax. The target of transition “terminateMeasurement” in the correspondence graph and all other irrelevant elements from all graphs are not shown.

are using other namespaces that are not part of the desired semantics. E.g. there may be several nodes of type “TypeDeclaration”, connected to a “SimpleType” via an edge of type “superInterfaceTypes”, but they are relevant for us only if the name of the “simple type” is “IState”.

4 Transformation rules

Rules are needed to insert and remove states, insert and remove transitions and insert or remove guards and updates to the state machine. From each of these TGG rules we can derive several operational rules. They are used to propagate changes of G_C to G_V and vice versa as well as to cover simultaneous transformations of all three graphs of the triple, making up a sum of six rules that can be derived from one TGG rule. For this paper, we derived the rules manually. The special use case of generating a complete verification model for given source code or generating source code for a given model can be understood as stepwise insertion of elements from scratch and is hence covered by the general rules for adding elements. Also the original use of Triple Graph Grammars as productions starting from an empty host graph is covered by this.

For example, the six rules for adding and removing a transition can be grouped to pairs according to the subsets of graph triple elements they use. Rules 1 and 2 are used to add and remove a transition simultaneously in both syntax graphs. Rule 3 is used for propagating a transition that has been added in G_C though G_M towards G_V. Inversely, rule 4 is used for propagation of deleting a transition the same way. Rules 5 and 6 do the same the other way round, from G_V via G_M towards G_C. One of these rules will be displayed and discussed in more detail in the following. Please note, that the use of auxiliary nodes or variables is skipped from these rules for clearness as well as the use of default attribute on the RHS, if a new node is created.

Another set of six rules is needed for adding and removing states. It is simpler than the one used for adding and removing transitions. In contrast to this, the rules for simultaneous manipulation and propagating changes in guards and updates are much more complex. Syntax trees for condition expressions may consist of virtually any number of syntactical elements. A series of rules has to applied after adding a condition to handle the expression stepwise. Removing conditions is simpler, because they can be disconnected from the transitions by deleting one edge and then be removed by a generic set of rules for removing the dangling subtree. Further rules are needed as well, for example to mark the initial state of the state machine or to compose a list of model variables. As stated before, rules must not destroy existing code structures but must be able to add default syntactical elements when implementing a state machine from scratch. Hence the rule set is complemented by rules to provide those code skeletons.

4.1 Sample Rule: Adding a transition simultaneously

We use the notation of the AGG tool environment for algebraic graph transformations [Tae00] in this section. Nodes and edges on the Right Hand Side (RHS) of a rule identified with nodes and edges from the Left Hand Side (LHS) of a rule are prefixed with numbers. Nodes that must not appear in a graph to get a matching are drawn as Negative Application Condition (NAC). Again nodes identified with nodes from the LHS are prefixed. String attributes are written in quotes, while variables used during the transformation are written without quotes.

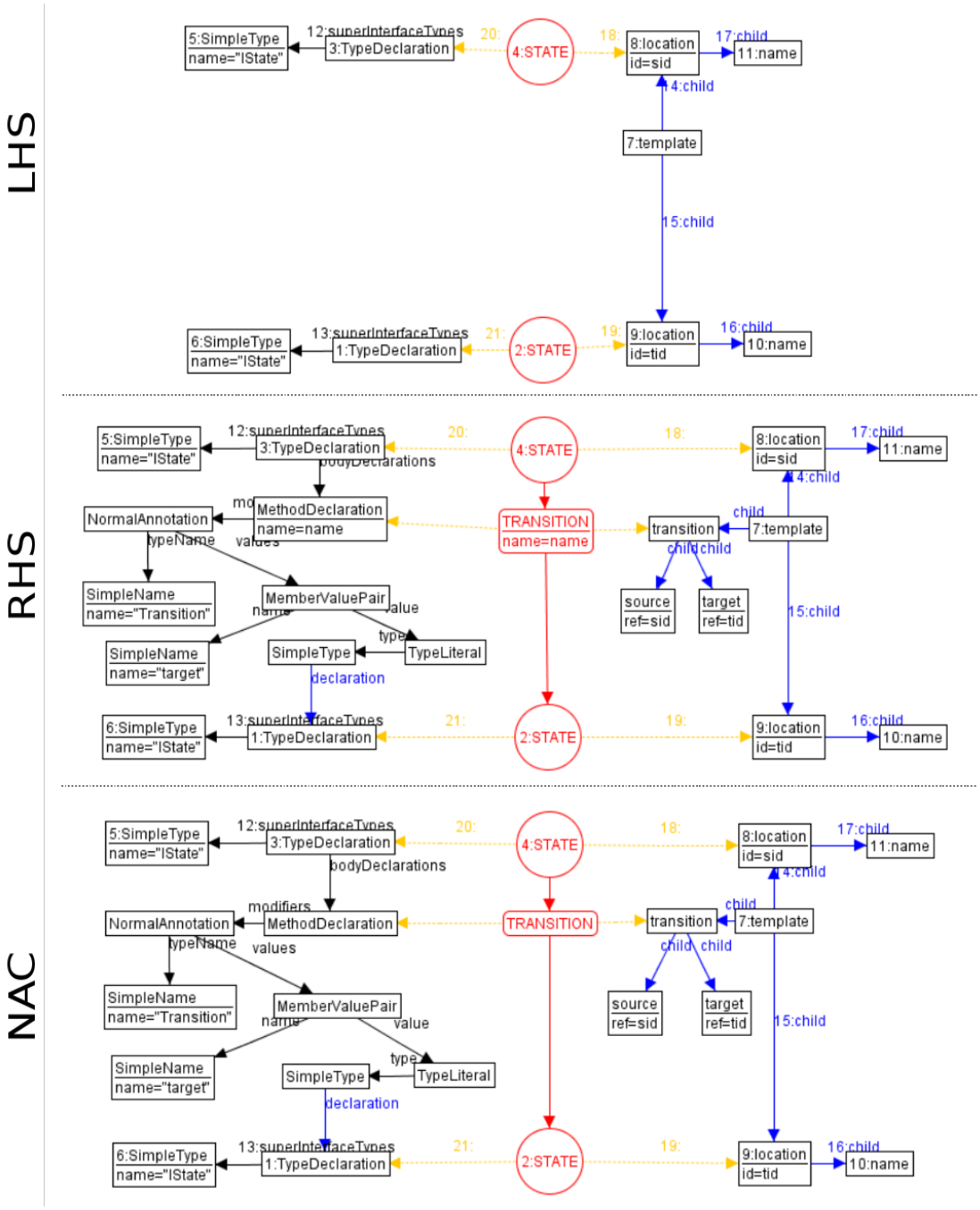


Figure 6: Sample rule for adding a transition simultaneously. Note that we are using variables “sid” and “tid” on the LHS to read attribute values from “location" nodes that are reused on the RHS in the “source” and “target” nodes. A value for variable “name” in the “TRANSITION” node on the RHS has to be provided externally.

The rule for simultaneous insertion of a transition in all graphs is shown in figure 6. As in figure 5, elements on the left half of the figure origin from G_C and elements on the right half origin from G_V. Nodes denoted by circles or rectangles with rounded corners origin from G_M. The LHS denotes a graph consisting of two states with complete mapping from G_M to G_C and G_V. No elements of transitions are relevant on the LHS. Moreover, the NAC explicitly enforces their absence. Note that elements from both syntax graphs are included in the NAC, to clearly distinguish this rule from other cases in which a transition may be already be added to one of the syntax graphs by manual manipulation. Furthermore, the transition node from G_M is also included in the NAC, because a situation with missing transitions in the syntax graphs but present transition in G_M would not be valid and hence out of the scope of this rule.

The RHS of the rule shows the completed RHS, including (1) the new "TRANSITION" node in the correspondence graph, connected to the existing state nodes by new edges, (2) a "transition" node and two adherent nodes "source" and "target" in G_V, connected to the existing node "template", and (3) several nodes in G_C for the programming language, i.e. a new method with some annotations in this example. It is assumed that the attributes "name" for the transition both in G_M and G_C can be set by a variable `name`. The value has to be provided by the user when triggering the rule application.

5 Implementation

Our approach has been implemented using version 1.6.2.2 of AGG. As mentioned in section 4, rules for AGG were derived manually. XML files were retrieved from their graph structure and written back by applying a XSLT provided at the AGG website [AGG]. For retrieving and writing syntax graphs for Java, a plugin for the Eclipse IDE was used, which is based on the Java Development Tools (JDT) [JDT08] as underlying API. This plugin is able to read and write syntax graphs for Java and make them accessible for the graph transformation API of AGG.

For reading source files, it uses the internal parser of Eclipse and accesses the resulting data structure. Each object in this structure represents one node in a syntax tree. Hence this structure is traversed and node objects in an AGG grammar are created for each object in the structure. Attributes of the graph nodes and edges are set according to the attributes of the objects in the data structure. During traversal of the object structure, additional information is collected in order to insert additional edges, extending the syntax tree to a syntax graph. Edges of type "access" used in section 3.1.2 are such additional edges. However, the plugin is designed for more general purposes and able to insert other types of edges.

For writing source files, an observer keeps track of changes triggered by applying rules and can map them back to the abstract syntax tree. As far as Eclipse does not allow to change the object structure directly, writing back the changes needs to replace the existing Java files by new ones generated from the graph. Obviously this is a major drawback and limits the practical use of the tool chain. In fact, truly simultaneous manipulations cannot be realized this way. However, extracting models from code and generating code from models is not affected by this limitation.

A variation of our concept is to divide each rule into two steps. The first step propagates a change from one syntax graph to the abstract model of the finite state machine and the second step propagates it to the other syntax graph. Thereby we reduce the number of nodes both on

LHS and RHS of the rules and make them less complex. Furthermore, it allows to use different rules in the second step and hence extend the graph triple to a n-tuple, supporting more than one specification or programming language. Of course the result is not a TGG any longer.

Based on this variation, we are able to use the approach to extract the embedded model semantics from given source code, even if they were not initially generated by graph transformations via the TGG. The rule set used to achieve this consists of 49 operational rules. Up to minor corrections, the complete tool chain from parsing source files, translating them to graphs, adding the correspondence graph, adding the XML graph and applying XSLT is successfully implemented in the plugin mentioned above.

6 Evaluation of the approach

Implementations of our approach are intended to be used during the design and implementation phase inside the software development process. While implementing the rules and doing practical experiments, we made several observations that will be discussed in the following.

6.1 Benefits

One major benefit of our approach is the bijectivity of the mapping. Thus it is irrelevant, which graph is changed by a programmer or modeller, because changes are triggered from both ends. Furthermore, rules preserve existing structures and can thus be used in cases, where parts of existing code or models have to be protected. The use of graph transformations is hence superior to code generating techniques, e.g. using code templates, because it is reversible and less obstructive regarding existing structures. Additionally, the bijectivity of the mapping allows it to track back errors directly from code to the model, allowing to use model checking techniques easily to analyse the implementation.

Another important point is the fact, that changes to the syntax graphs can be triggered by other means than graph transformation. This makes it possible to integrate the approach into tools working on graphs but without means of graph transformations. In fact, editing will not be simultaneous as with synchronous rule application in this case, but rule matching can be done continuously and will automatically catch every relevant change.

Proofs for completeness and correctness are not considered here, but since the rules can be derived systematically from the mapping, correctness could in principle be proven by showing the correctness of both the mapping and the derivation process. This makes the approach extendable without need for new explicit proof obligations. Additionally, the rules derived from the Triple Graph Grammar can be used to prove the implementation of a model to be correct. Tools designed for the use of TGGs can help to automate this process.

6.2 Drawbacks

Memory structures of editing systems are not necessarily equal to their data format representation in an abstract syntax tree. Additionally, object structures representing this abstract syntax tree inside editing systems may be graph-like, but not prepared for the application of graph transformations. Both facts make it necessary to export these structures into a graph format suitable

for graph transformation and write them back afterwards. The result is a significant slowdown and makes it difficult to realize simultaneous editing. As mentioned in section 5, tool integration of simultaneous editing is not possible at the moment. The general problem of read and write operations is not limited to our approach or the special case of tool integration, but also discussed in [JKS06]. Moreover, imports and exports may be error-prone, so formal proofs are depend on the correctness of the import/export routines as well.

While the abstract model expressed in G_M is able to cover all semantics of a finite state machine, this is not necessarily true for the syntax of the chosen specification or programming language. Hence the mapping from G_M can only be complete for the least common subset of supported operations in the chosen languages.

6.3 Future Work

One of the next goals is stronger and more direct tool support, using internal data structures of editors to get rid of the need for explicit reading and writing files. This implies the need of direct access to an API for tools handling specification languages. This would make the use of XSLT superfluous and could integrate everything into one tool. Additionally, when using an API there would be not longer the need to write the graph structure explicitly to disk. A better performance can be expected if this step can be skipped. In an optimal solution an IDE would integrate source code editor, model editor and graph transformation engine, all sharing the same graph base data structure. The vision is to allow both truly simultaneous and virtually simultaneous editing this way. In truly simultaneous editing, the user marks matching points either in the code or the verification model and applies rules directly. In virtually simultaneous editing, a background process checks for changes permanently and every change triggers the rule application and is propagated thereby through the models.

Our general approach is not limited to the use case of state machine implementations. In theory it can be used in any case in which a model can be mapped directly to an implementation. Considering Java as a platform independent language and thus omitting the need of a platform specific model in the traditional model-driven development process, this enables direct bidirectional connections between various kinds of platform independent models and the whole application source code. However, the approach presented in this paper is no general approach to map any arbitrary code structure to a related model. The key requirement for generating TGG rules and deriving transformation rules is to know how a model is mapped to a programming language and which syntactical elements are used for this mapping.

7 Conclusions

In this paper, we presented a Triple Graph Grammar to connect finite state machine models with their implementation in source code. We explained the explicit mapping for states and transitions and showed how to derive rules for simultaneous manipulation and bijective change propagation from this by example. The derivation followed a schema with equal or inverse parts in the rules and can hence be used as a general process. We discussed the benefits of this approach and the drawbacks existing in the current implementation.

It can be summarized, that the approach allows the true simultaneous manipulation of two graphs as well as the propagation of changes from one syntax graph to another. While this is no new result for TGGs in general, it is new for the relation between models and source code. Hence classical model transformation and code generation from model-driven development can be replaced or at least supplemented by manipulations and transformations based on a bijective mapping. This way, we gain better possibilities for verifying implementations and tracking back errors to the models.

Bibliography

[ABC07] M. Antkiewicz, T. T. Bartolomei, K. Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. Pp. 214–223. ACM, New York, NY, USA, 2007. doi:http://doi.acm.org/10.1145/1321631.1321664

[AC06] M. Antkiewicz, K. Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. In Nierstrasz et al. (eds.), *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*. Lecture Notes in Computer Science 4199, pp. 692–706. Springer, 2006.

[AGG] AGG website. http://tfs.cs.tu-berlin.de/agg/.

[BEK+06] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, E. Weiss. EMF Model Refactoring based on Graph Transformation Concepts. In *Proceedings of Third International Workshop on Software Evolution through Transformations (SETra'06)*. Volume 3. Natal, Brazil, sept 2006. Electronic Communications of the EASST.

[BLW05] P. Baker, S. Loh, F. Weil. Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. Pp. 476–491 in [BW05].

[BSG08] M. Balz, M. Striewe, M. Goedicke. Embedding State Machine Models in Object-Oriented Source Code. In *Proceedings of the 3rd Workshop on Models@run.time at MODELS 2008*. Pp. 6–15. 2008.

[BSG09] M. Balz, M. Striewe, M. Goedicke. Embedding Behavioral Models into Object-Oriented Source Code. In *Software Engineering 2009. Fachtagung des GI-Fachbereichs Softwaretechnik, 2.-6.3.2009 in Kaiserslautern*. 2009.

[BW05] L. C. Briand, C. Williams (eds.). *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*. Lecture Notes in Computer Science 3713. Springer, 2005.

[CEM+06] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (eds.). *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande*

do Norte, Brazil, September 17-23, 2006, Proceedings. Lecture Notes in Computer Science 4178. Springer, 2006.

[GL06] E. Guerra, J. de Lara. Model View Management with Triple Graph Transformation Systems. Pp. 351–366 in [CEM+06].

[GW06] H. Giese, R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6*. Pp. 543–557. 2006.

[HT06] B. Hailpern, P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal* 45(3):451–461, 2006.

[JDT08] Eclipse Java Development Tools. 2008. http://www.eclipse.org/jdt/.

[JKS06] J. Jakob, A. Königs, A. Schürr. Non-materialized Model View Specification with Triple Graph Grammars. Pp. 321–335 in [CEM+06].

[JZ99] J. H. Jahnke, A. Zündorf. *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 2, chapter Applying Graph transformations to database reengineering. World Scientific, 1999.

[Kö05] A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS 2005*. Montego Bay, Jamaica, 2005.

[LPY97] K. G. Larsen, P. Pettersson, W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer* 1(1–2):134–152, Oct 1997.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Mayr et al. (eds.), *Graph-Theoretic Concepts in Computer Science*. LNCS 903. 1994.

[SK04] S. Sendall, J. Küster. Taming Model Round-Trip Engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*. 2004.

[Tae00] G. Taentzer. AGG: A tool environment for algebraic graph transformation. In Nagel et al. (eds.), *Application of Graph Transformation with Industrial Relevance: International Workshop, AGTIVE'99*. Lecture Notes on Computer Science 1779, pp. 481–488. Springer, Kerkrade, The Netherlands, 2000.

[Tae08] G. Taentzer. Construction of Consistent Models in Model-Driven Software Development. In Kutsche and Milanovic (eds.), *Model-Based Software and Data Integration, Frist International Workshop, MBSDI 2008, Berlin, Germany, April 2008*. Communications in Computer and Information Science 8, pp. 113–124. Springer, 2008.

[WS05] H. Wada, J. Suzuki. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. Pp. 584–600 in [BW05].