Electronic Communications of the EASST
Volume X (2010)

Proceedings of the
Fourth International Workshop on
Graph-Based Tools
(GraBaTs 2010)

Enabling Graph Transformations on Program Code

Michael Striewe, Moritz Balz, and Michael Goedicke

12 pages

Guest Editors: Juan de Lara, Daniel Varro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Enabling Graph Transformations on Program Code

Michael Striewe, Moritz Balz, and Michael Goedicke

Paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen, Essen, Germany
{michael.striewe, moritz.balz, michael.goedicke}@s3.uni-due.de

Abstract: Although the internal representation of program code in parsers and compilers is the abstract syntax tree and thus a graph, tools for handling program code as an explicit graph are rare. This contribution introduces a tool that generates abstract syntax graphs out of Java program code. Code can be read and stored as a graph, and code can be manipulated by the application of graph transformations. We show by examples how this can be used for low-level analysis and manipulation as well as for code interpretation at different levels of abstraction with formal models.

Keywords: Abstract Syntax Graph, Code Manipulation, Model Transformation, Graph Pattern Matching

1 Introduction

Program code is usually represented in a textual notation. Consequently, any manipulation on it can be expressed in terms of creating, replacing, moving, or deleting strings. However, the internal representation of program code as used by parsers and compilers is the notation of an abstract syntax tree, which is based on the type graph given by the grammar of the respective programming language. Reverse engineering and refactoring tools usually traverse this data structure for their manipulations. Since traversing is only one limited possibility to handle a graph, it is appealing to make this graph structures *explicitly* available for a general-purpose graph transformation engine. By this means any kind of code manipulation can be expressed as graph transformation rules. This is of interest whenever program code is to be analyzed, manipulated, or transformed according to well-defined rules.

In this contribution we present approaches that enable the use of graph transformation techniques for program code written in the programming language Java. Code can be read and stored as a graph, and code can be manipulated by the application of graph transformations. This contribution is organized as follows: Section 2 describes JAVA2GGX, which is the actual tool that makes Java syntax graphs available to a graph transformation engine. Sections 3 and 4 discuss low-level and high-level transformations of program code based on examples. The remaining two sections provide a brief overview about related work and draw conclusions.

2 The Basics: JAVA2GGX

In order to make Java source code available for graph transformation techniques, the explicit generation of syntax graphs is necessary. Our graph-based tools for manipulating and analyzing

```
public class SampleClass {
  public static void main(String[] args) {
    new AnotherClass(1).talk();
  }
}
```

Listing 1: Source code of SampleClass.java used as example in this section.

```
public class AnotherClass {
  public AnotherClass(int i) { }

  public void talk() {
    System.out.println("Hello␣world!");
  }
}
```

Listing 2: Source code of AnotherClass.java used as example in this section.

Java code are based on a component named JAVA2GGX, which in turn is based on AGG [AGG] and its file format GGX. In our case, JAVA2GGX uses the AGG programming interface (version 1.6.4) as an underlying engine for all graph transformation activities.

Abstract syntax trees are generated by a parser while processing source code. They reflect the structural elements of the source code using the grammar of a certain programming language. JAVA2GGX is capable of parsing Java files to their syntax tree according to the Java Language Specification for Java 6. Each structural element is represented by a node which is of a certain type and which possibly has attributes. Each connection between nodes is made up by directed arcs which are also typed and may have attributes.

Although an abstract syntax tree reflects all structural properties of source code, it is not always comfortable to work with a plain abstract syntax tree. For this reason JAVA2GGX automatically performs some modifications during parsing. These modifications introduce both new attributes and new arcs which implies that the result is no longer a tree but a graph. All modifications are explained in detail in the following.

An example of source code and the extracted graph is shown in listings 1 and 2 and figure 1. The layout has been done manually in AGG. Names for node types are identical to the class names used inside the parser for representing the abstract syntax tree. Arcs in blue color are additional arcs introduced by JAVA2GGX and thus not part of the parsed abstract syntax tree. They will not be considered if code is generated or rewritten from a graph.

2.1 Simplified Naming

According to the Java language specification, names are assigned to elements by referring to an element of type `SimpleName` or `QualifiedName`. In the syntax tree this is represented by outgoing arcs of type `name` towards nodes of type `SimpleName` or `QualifiedName` respectively, which contain an attribute for the name. While this structure is reasonable for language design, it adds an overhead of nodes and arcs to the graph. For this reason the graph is

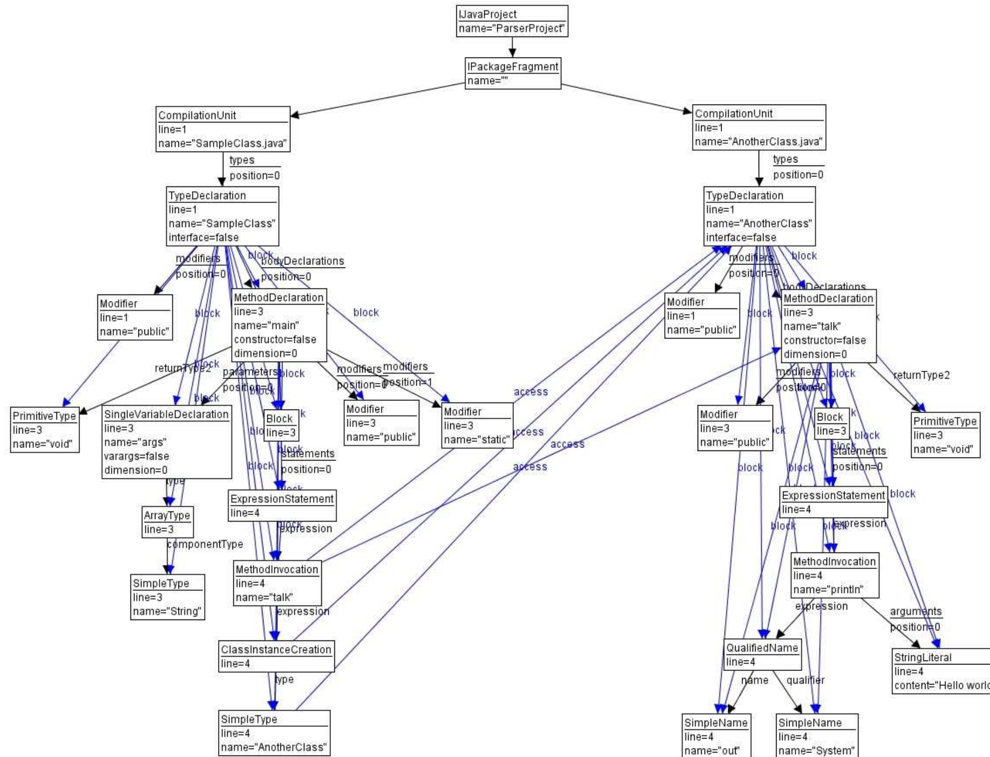


Figure 1: Sample graph generated by JAVA2GGX from the source code shown in listings 1 and 2.

simplified by assigning name attributes to several types of nodes directly, thus omitting the extra nodes for this purpose. In detail, this applies to the following structures:

- Nodes of type `QualifiedName` are omitted if they own the name property for nodes of type `ImportDeclaration` or `PackageDeclaration`. Instead, an attribute `name` is given to these nodes directly.
- Nodes of type `SimpleName` are omitted if they own the name property for nodes of type `MethodDeclaration`, `MethodInvocation`, `PackageDeclaration`, `SimpleType`, `SingleVariableDeclaration`, `TypeDeclaration` or `VariableDeclarationFragment`. Instead, these nodes are given an attribute `name` directly.

2.2 Line Numbers

Since the parser used inside JAVA2GGX provides information about the line number of each structural element, this information is included into each node. Thus each node owns an additional attribute containing the line number that is the origin of this element inside the respective source code file.

2.3 Blocks

Some Java constructs like methods or loops have a so-called “body” that is a block of statements. More precise, elements that span out a block have either outgoing arcs of type `bodyDeclarations` to each declaration statement inside the block or an outgoing arc of type `body` to an explicit `Block` node. By following arcs reverse to their direction until reaching arcs of the types named above, a path to the roots of blocks for each element can be found.

However, AGG does not support expressing paths in rules. In order to enable rules that consider block structures, JAVA2GGX introduces additional arcs of type `block`. They are drawn from a node s to a node t if there is a path from s to t where the first arc on the path is of type `body` or `bodyDeclarations`.

The example in figure 1 has five elements spanning out a block: Two type declarations and three method declarations. All elements inside these blocks are connected to their root elements by an additional blue arc of type `block`. Note that class modifiers are not part of the block and hence not connected by an additional arc. The same applies to method modifiers: They are not connected to the method declaration node by a blue arc, but belong to the elements inside the block of the surrounding type and hence have an incoming arc from there. Finally, elements that belong to the block of a method have two incoming arcs of type `block`, one from the method declaration and one from the surrounding type declaration.

2.4 Access Arcs

During parsing, names are resolved to references between syntactical elements. The parser assigns a unique identifier to each declaration of a type, method, or variable. Consequently, each call to a type, method, or variable uses this unique identifier as reference. This way the parser detects missing or duplicate declarations as well as global variables hidden by local declarations. JAVA2GGX displays these references between elements by introducing additional arcs of type `access`. There are several situations in which these arcs are drawn:

- A node of type `ClassInstanceCreation` is considered as a source of an additional arc that points to a node of type `MethodDeclaration` (which has its attribute `constructor` set to `true`) if the class instance creation uses a constructor explicitly given in the source code.

- A node of type `MethodInvocation` is considered a source of up to three additional arcs that point to a node of type `MethodDeclaration`, its parenting `TypeDeclaration`, and the related `PackageDeclaration`. The latter are only drawn if they do not point to parents of the `MethodInvocation`. However, a reference from `MethodInvocation` to `MethodDeclaration` is always drawn, thus possibly indicating a recursive method. In addition, the parenting `TypeDeclaration` of the `MethodInvocation` is also considered a source of another arc of type `access` pointing to the parenting `TypeDeclaration` of the `MethodDeclaration`. Thus types referencing each other can be identified without considering all their declared methods.

- A node of type SimpleName is considered a source of up to three additional arcs. Since nodes of type SimpleName can occur in different situations, three cases have to be distinguished:
  - If the name refers to a type, two arcs may point to the TypeDeclaration and the related PackageDeclaration, but are only drawn if they do not point to parents of the SimpleName itself.
  - If the name refers to a class member variable, one arc points to the related SingleVariableDeclaration. Another arc points to the parenting TypeDeclaration. A third arc points to the related PackageDeclaration if it is not the same as the one related to the SimpleName itself. In addition, the parenting TypeDeclaration of the SimpleName is also considered a source of another arc of type access pointing to the parenting TypeDeclaration of the SingleVariableDeclaration.
  - If the name refers to a local variable of a method, a single arc points to the related SingleVariableDeclaration or VariableDeclarationFragment.
  - If the name refers to an enumeration constant, a single arc points to the related EnumConstantDeclaration.
- A node of type SimpleType is considered a source of an arc pointing to the related TypeDeclaration. Another arc points to the related PackageDeclaration if it is not the same as the one related to the SimpleType itself. In addition, the parenting TypeDeclaration of the SimpleType is also considered a source of another arc of type access also pointing to the TypeDeclaration.

In figure 1 one can see five blue arcs of type access. All are pointing from the left subgraph to the right subgraph. The arc most easy to understand is the one from methodInvocation on the left to methodDeclaration on the right. Obviously the method “talk()” called in line 4 of SampleClass.java is the one declared in line 6 of AnotherClass.java, which is expressed by this arc. As explained above, a node of type MethodInvocation may be source of up to three arcs. Another one is visible in the example, pointing towards a node of type TypeDeclaration. As one can see, this node represents the type in which the method “talk()” is declared.

Another arc of type access leads from a node of type ClassInstanceCreation to a node of type MethodDeclaration. In this case the target is the constructor used for this class instance creation as declared in line 3 of AnotherClass.java. The type used in this class instance creation is attached to the node of type ClassInstanceCreation as another node of type SimpleType. This node also has an outgoing arc pointing to the node of type TypeDeclaration introduced above. In this case the arc means that in line 4 of SampleClass.java a class is instantiated which is declared in line 3 of AnotherClass.java.

Up to here, all arcs started at nodes representing a certain statement with a direct reference to a method or type. As explained above, parenting nodes are also considered by JAVA2GGX. Hence a fifth arc of type access exists in figure 1. It leads from the node of type TypeDeclaration on the left to another node of same type on the right. It is introduced since some elements in

the subtree on the left side access either the node on the right directly or access elements from its subtree. Hence by only looking at this arc you can already see that the type “SampleClass” accesses “AnotherClass” in some way.

2.5 Termination Conditions

In loops of type “for”, “while”, and “do ... while” there is an expression providing an explicit termination condition. In the AST nodes of type `ForStatement`, `WhileStatement`, and `DoStatement` have an outgoing arc of type `expression` leading to the root of the subtree representing the termination condition. To gain explicit information about nodes used in this subtree, additional arcs of type `termination` are introduced by JAVA2GGX during parsing for all nodes inside this subtree including the root itself. Thus, whenever there is a path from a node s of type `ForStatement`, `WhileStatement`, or `DoStatement` to another node t with the first arc of the path being of type `expression`, there will be an additional blue arc of type `termination` leading from s to t.

3 Low-level Transformations and Analysis

As the abstract syntax graph is a detailed representation of software, it allows for transformations and analysis on the level of single source code elements and small patterns of few elements. We refer to these as low-level transformations and analysis in the following, in contrast to high-level transformations and analysis related to abstract specifications which we will discuss in section 4.

The goal of low-level analysis of program code as discussed in this paper is to determine whether certain simple constructs expressed by patterns are present or absent in the given source code. This allows to check for style and programming conventions, e.g. searching for empty `catch` clauses or unused method parameters. Since this is known as “static analysis” of source code, it is no particular feature of JAVA2GGX, but can also be done with many other typical tools for static code analysis like CHECKSTYLE [Che] or PMD [PMD]. These tools also use the abstract syntax graph. However, they do not handle graphs as an explicit data structure, but traverse the AST programmatically. Contrary, patterns to be searched by JAVA2GGX have to be defined as graph transformation rules instead of relying on programming, thus taking advantage of the descriptive rules. Since these rules cannot be implemented straightforward, one of the following strategies is to be used:

- Optimistic rules assume that a given piece of code is correct unless an undesirable structure is present. In this case, the erroneous structure is placed on the left hand side (LHS) of the rule. The right hand side (RHS) contains the same structure since changes to the code are not intended. An additional “error node” is added there, which is a node of a fixed type that does not appear in normal syntax graphs and that contains a message describing the detected error. So whenever the erroneous structure is present in a solution, this rule matches and inserts the error node into the graph.

- Pessimistic rules assume that a given piece of code is wrong unless a certain structure is present. The LHS of a rule is empty (so that it is always applicable) and correct structures

are added as negative application conditions, preventing the rule from being applied if they are found in the graph. The RHS contains only an appropriate error node.

While this seems to be much more complicated on the first glance, it allows for direct combination of pattern search and program manipulation. With JAVA2GGX, spotted undesirable patterns can be changed based on transformation rules directly, or pattern search can be performed by the application of alignment rules. These rules may adjust syntactical divergences before rules for searching special patterns are used. This is interesting in contexts where not one large software project is analyzed, but where many similar but different projects are searched, i.e. in e-learning and automated assessment [KG06, KG08]. In addition, the explicit use of error nodes as markers allows to store results of a search inside the graph data structure and reuse them later on with more complex rules. If source code is to be generated out of manipulated syntax graphs, any additional nodes and arcs introduced by JAVA2GGX are ignored and only nodes belonging to an abstract syntax tree are used to produce the textual representation.

4 High-level Transformations and Analysis

So far we considered analysis and manipulation of Java code at the level of programming language constructs. While this is helpful in the cases introduced so far, source code is often related to abstract specifications like formal models. Transformations for them are already used in model-driven software development [BEK+06, JKS06]. However, an issue considered in this context is that program code is usually derived from specifications unidirectionally with code generation [HT06] only. The reading and writing access to Java code for graph transformations allows for more sophisticated approaches of integrating model specifications and program code since the code can fully participate in such transformations.

4.1 Approach

The reason for fact that (graph) transformations considering source code are usually unidirectional is that generated code does no longer contain the semantics of abstract models. Source code can thus only be integrated in bidirectional transformations if it follows certain rules that make an interpretation possible based on unambiguous rules. To overcome the gap between abstract specifications of software and implementations that are partly created manually instead of being completely generated out of models, we proposed the approach of *embedded models* [BSG10]. An embedded models defines a program code pattern that carries the abstract syntax of a formal model. The pattern code is then not used as meta data, but is interpreted and executed at run time by an execution framework that realizes the execution semantics of the underlying formal model. The pattern code is connected to arbitrary other program code by means of interface code used for data exchange and initiation of business logic execution.

The architecture of embedded models is illustrated in figure 2: The program code pattern is integrated in other program code by invocations of interface code and provision of entry points used by the execution framework. At the same time, transformations exist that use the embedded model syntax to connect the program code to abstract specifications. By this means the source code can be integrated in the chain of transformations at the model level.

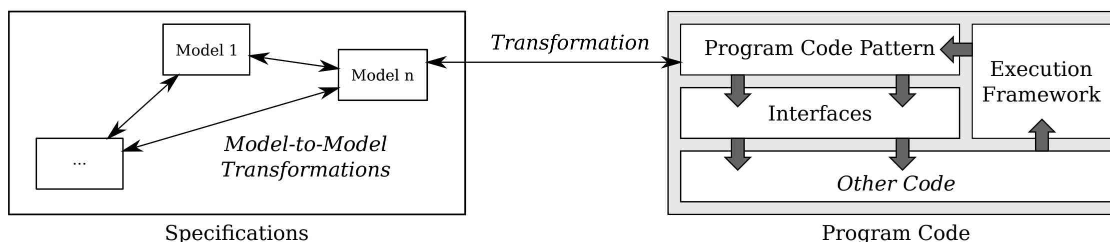


Figure 2: The elements of an embedded model definition. The program code following the pattern can be interpreted unambiguously so that it can be integrated in model transformations across different abstraction levels.

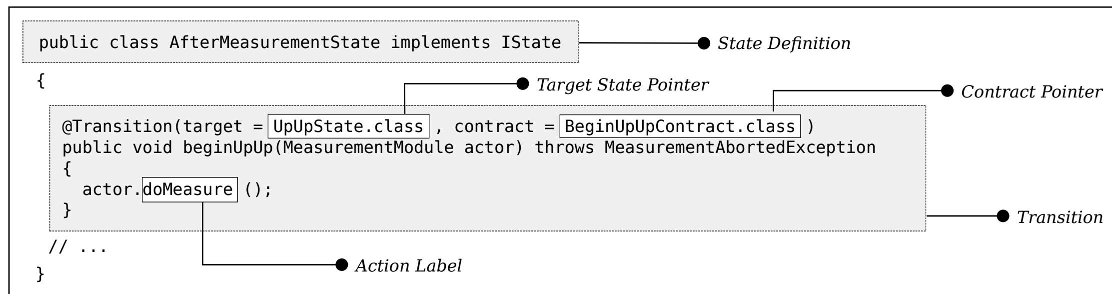
One of the embedded models developed so far considers state machine models. Basically, these models are constituted by a set of classes implementing states and transition contracts, while transitions are implemented as methods. Figure 3 explains some details: The class at the top implements a marker interface `IState`, so it is a state class whose unique name represents the state's name. Its methods are marked as transitions by a Java meta data annotation `@Transition` whose attributes refer to the target state and a contract class (bottom of figure 3) containing guards and updates. An interface type referred to as "actor" is passed to transition methods. Its methods are interpreted as action labels which are called when the transition fires.

Guards and updates are implemented as two methods in a contract class, both evaluating boolean expressions. The method `checkCondition` acts as a guard, deciding whether the related transition may fire or not. It uses the current variable values of the state machine for this decision. The method `validate` acts as an update validator, indicating whether variables match the expected value or value range after actions have been performed. Thus it compares the current values with the values from the point in time before the transition fired. Both methods access a "variables" type which is a facade type representing the variables constituting the state space of the state machine.

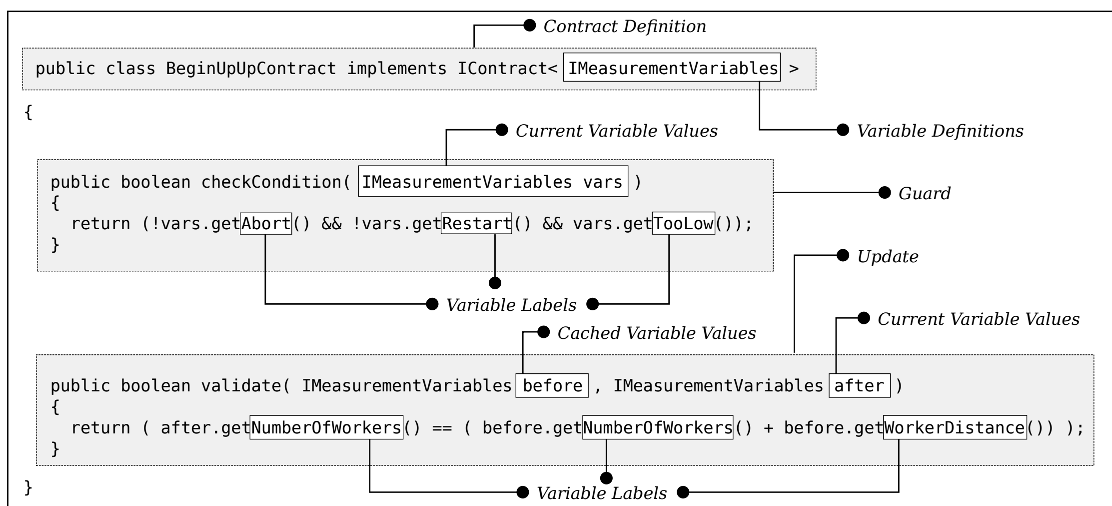
4.2 Program Evolution by Model Transformation

The use of embedded models allows to consider program code at higher levels of abstraction and thus to apply high-level transformations, too. As stated above, state machine models are only one example for embedded models. Thus we will now discuss an example in which we use both state machine models as well as process models [BG09]. At the level of models a state machine can be transformed into a process model automatically while losing only just a few features of state machines that cannot be expressed in process models. Since embedded models rely on static structures in program code and JAVA2GGX allows to transform these program code structures into a graph, model transformation rules that are able to transform a state machine model into a process model or vice versa can be rewritten in order to transform program code with an embedded state machine into program code with an embedded process model or vice versa respectively. For a detailed description please refer to our previous publication [GSB09].

The actual set of graph transformation rules used to implement the evolution from state machine models to process models consists of 21 rules. At first, two rules are concerned with



State and Transition Definition in Source Code



Contract Definition in Source Code

Figure 3: A state definition with an outgoing transition and its contract. The first method of the contract checks a pre-condition with the current variable values, while the second method checks a post-condition by comparing the current values to previous values.

converting states to decision nodes and transitions to activity nodes. One of these rules – changing states to process nodes and creating activity nodes – is shown in figure 4 in a simplified manner. Due to the use of embedded models, elements to be moved can easily be identified by their annotations on the left hand side of the rule and thus reassembled on the right hand side. Similarities between state machines and process models allow to reuse larger parts of existing program code, e.g. complete method bodies.

After nodes have been converted or created, a set of six rules applies changes for cleaning up the graph, like reordering imports or removing unnecessary annotations. These rules are not necessary for a valid embedded process model, but they remove unused nodes, thus helping to keep the graph small. Two additional rules remove contracts not longer needed as well as useless decision nodes that have been introduced by rules applied earlier. Afterwards, the set of nodes is complete, so three rules can be applied for marking start nodes, end nodes, and merge nodes. An additional rule is necessary for splitting up activity nodes since an embedded state machine allows to have more than one action label in transitions, but only one activity per activity node is

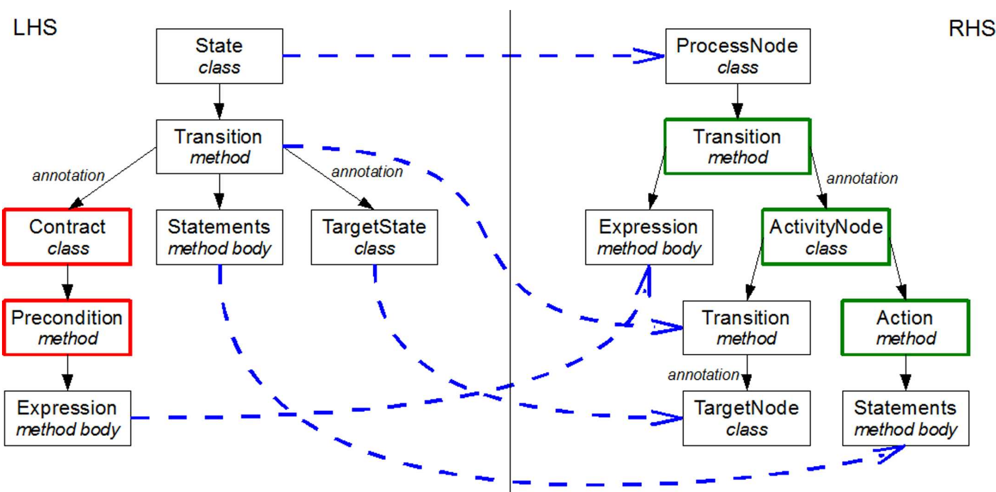


Figure 4: Simplified graph transformation rule for transforming states into process nodes. Nodes deleted from the syntax graph are marked in red while newly created nodes are marked in green. Some of the preserved nodes are renamed during transformation. Note how contents from the original state node are moved to a newly created activity node, while contents from the original extra contract node are moved into the existing process node.

allowed in embedded process models. Another set of seven rules is finally concerned with some adjustments to the code.

For a sample instance of a state machine containing 5 state classes and 5 contract classes, processing the 10 files with JAVA2GGX resulted in a graph with 331 nodes and 694 arcs. Transformation was finished after a total amount of 131 rule applications. Most applications where used for housekeeping on the graph (e.g. 36 for correcting “block” arcs, 32 for copying import statements and 14 for removing unused imports). On an Intel Core2 Duo CPU at 3.17 GHz and with 4 GB RAM the transformation took about 10 seconds.

4.3 Transformation for State Machine Verification

The transformations presented in the previous subsection can be called “internal” transformations since they consider only program code and its abstract syntax tree. However, we can of course also think of “external” transformations, where the abstract syntax tree of the program is part of a larger context. This is obviously the case if source code is derived from models by a sequence of transformation or generation steps. However, this larger context may also be a triple graph grammar, where the syntax graph is one element of the triple. The second element of the triple can be the representation of state machines as used by a verification tool (e.g. UPPAAL [LPY97]) as we have already shown [Str08]. Insufficient graph manipulating capabilities of UPPAAL do not allow to realize synchronous manipulations of models in UPPAAL and embedded models in source code, but transformation rules allow to propagate changes trough the triple graph structure in both directions.

5 Related Work

The principles of graph transformations for program manipulations have already been discussed some years ago in the context of refactoring [EJ04]. Tools like SPOON [PNP06] realize this based on graph traversals without providing the graph explicitly. More recent tools focus on meta modeling and integration into the Eclipse Modeling Framework (EMF) – e.g. JAMOPP [HJSW09] or the MODISCO project [MoD] – and thus come close to the high-level transformations described above in a more abstract way. For UML diagrams, such transformations have also been realized for AGG [FM07].

With JGRALAB it is possible to transform java source code into so-called TGraphs which can be used for declarative graph queries with a query language named GReQL [JGr]. While this allows for detailed program analysis based on graphs, it does not provide means for transformations of program code. The same applies to tools for static code analysis (like PMD as already mentioned above). They use syntax graphs or trees and allow to specify patterns to be searched, but do not make the graphs generally accessible for transformations.

6 Conclusion

In this contribution we introduced JAVA2GGX, a tool for generating explicit graph representations out of abstract syntax graphs of Java programs and vice versa. Using these graphs it is possible to apply static program analysis and low-level transformations for refactoring using graph transformation rules. The benefits are that any means known from graph transformations (e.g. use of additional node types as markers) can be applied to these problems this way. In addition, high-level transformations are also possible if appropriate code structures (e.g. embedded models) are used. By this means another benefit for software evolution is achieved, which is a tighter integration of model transformation and code transformation.

Future work is necessary with respect to content as well as to implementation. At the time of writing, JAVA2GGX can be used as a standalone tool and as a plugin for the development environment Eclipse. This is sufficient for internal transformations, but not for external ones. With respect to content, many other transformations can be developed than the ones shown in this contribution, for different purposes like analysis, verification, and evolution of software.

Bibliography

[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.

[BG09] M. Balz, M. Goedicke. Embedding Process Models in Object-Oriented Program Code. In *Proceedings of the First Workshop on Behavioural Modelling in Model-Driven Architecture (BM-MDA)*. 2009.

[BSG10] M. Balz, M. Striewe, M. Goedicke. Continuous Maintenance of Multiple Abstraction Levels in Program Code. In *Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development - FTMDD 2010, Funchal, Portugal*. 2010.

[Che] CheckStyle Project. http://checkstyle.sourceforge.net.

[EJ04] N. van Eetvelde, D. Jannsens. Extending graph rewriting for refactoring. In *Proceedings of International Conference of Graph Transformation (ICGT) 2004*. Springer, 9 2004.

[FM07] A. Folli, T. Mens. Refactoring of UML models using AGG. *ECEASST* 8, 2007.

[GSB09] M. Goedicke, M. Striewe, M. Balz. Support for Evolution of Software Systems using Embedded Models. In *Design for Future – Langlebige Softwaresysteme*. 2009.

[HJSW09] F. Heidenreich, J. Johannes, M. Seifert, C. Wende. JaMoPP: The Java Model Parser and Printer. Technical report, Technische Universität Dresden, 2009.

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

[JGr] JGraLab. http://userpages.uni-koblenz.de/~ist/JGraLab.

[JKS06] J. Jakob, A. Königs, A. Schürr. Non-materialized Model View Specification with Triple Graph Grammars. In Corradini et al. (eds.), *Proceedings of the 3rd International Conference on Graph Transformation (ICGT) 2006, Natal*. Lecture Notes in Computer Science 4178, pp. 321–335. Springer, 2006.

[KG06] C. Köllmann, M. Goedicke. Automation of Java Code Analysis for Programming Exercises. In *Proceedings of the Third International Workshop on Graph Based Tools*. Electronic Communications of the EASST 1. 2006.

[KG08] C. Köllmann, M. Goedicke. A Specification Language for Static Analysis of Student Exercises. In *Proceedings of the International Conference on Automated Software Engineering*. 2008.

[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.

[MoD] MoDisco Project. http://www.eclipse.org/MoDisco/.

[PMD] PMD Project. http://pmd.sourceforge.net/.

[PNP06] R. Pawlak, C. Noguera, N. Petitprez. Spoon: Program Analysis and Transformation in Java. Technical report 5901, INRIA, 2006.

[Str08] M. Striewe. Using a Triple Graph Grammar for State Machine Implementations. In Ehrig et al. (eds.), *Proceedings of the 4th International Conference on Graph Transformations (ICGT) 2008, Leicester*. LNCS 5214, pp. 514–516. 2008.