Automatic Generation of Graph Models for Model Checking

E.J. Smulders
University of Twente
edwin.smulders@gmail.com

ABSTRACT

There exist many methods to prove the correctness of applications and verify the source code. GROOVE (GRaphs for Object-Oriented VErification) is a tool that can do such verifications, but the input graph models have to be created by hand. Currently, there exist some tools to generate models from Java source code, such as Bandera, but these tools do not make it possible to add other languages as input, nor do they generate GROOVE models (graph transformations).

This research is part of a project to create a generic verification tool. This tool takes an ANTLR grammar as the syntax definition for the language used and source-to-model transformation rules which take into account the semantics of the language. With these rules, a GROOVE model is generated. Using this, one can verify any source code for which a grammar and transformation rules are available.

This research focuses on the creation of the actual transformation rules to generate a model that resembles the transformation graphs to be used with GROOVE. This is done with Gra2MoL, a rule language to generate models from a grammar. The focus lies on getting working rules for a basic subset of Java, such as assignments and simple conditional branching. In this paper we present the results, along with some shortcomings and drawbacks.

Keywords

GROOVE, graph transformations, code verification, model checking, model extraction, Java, Gra2MoL

1. INTRODUCTION

Proving the correctness is an important part of application validation. To do this, you need to verify that the program conforms to the requirements. There exist many tools to do model checking on a pre-defined model. GROOVE (GRaphs for Object-Oriented VErification) is such a tool for verification of object oriented applications[4]. It uses graphs to represent a state in the application and graph transformations to modify this state. With these graphs, it can calculate the state space and, if you specify error states, check if these error states are reachable. There is however one problem with using GROOVE; the graph-based models have to be created manually from the source code, which is not trivial for large applications and error-prone (resulting in incorrect verification results). In this research, we want to automate the generation of the graph models needed for GROOVE.

The project scope defines an extra requirement. In some programming languages, there exists a language construct called a macro, which embeds one language into another. Good examples of such combined languages are the embedding of assembly in C and writing SQL statements in a string (in any language). Because these are often very different languages, it poses a problem for verification tools, which often verify only one language. The research explained in this paper falls within the scope of a project to create a tool which handle multiple languages in a generic way. This will be done by combining ANTLR (ANother Tool for Language Recognition) grammars[1]. Because of the relation to grammar of the project scope, Gra2MoL (Grammar to Model Language)[2] is a good candidate as a language to extract the models from the source code.

The following section contains the background information to the research. Section 3 contains the related work. Section 4 defines the problem statement. In section 5, the approach is outlined. In section 6, we continue with the results and challenges. We finalize with the conclusions in section 7 and the references (section 8).

2. BACKGROUND

2.1 GROOVE

GROOVE uses a state graph combined with transformation graphs to verify the states a program can reach. Currently, these graphs have to be created manually.

Figure 1. Example of a GROOVE model start state

Figure 1 is an example of a (start) state of a GROOVE model. The nodes in such a state represent an object in the application the model represents. Edges represent a reference to another object. To a state like this, trans-
formations are applied through transformation graphs. In figure 2 an example of a GROOVE transformation graph can be seen. It contains colors, which have been clarified in the description of the nodes and transitions.

Figure 2. Example of a GROOVE transformation graph

- The black edges and nodes (next, Cell and Buffer in the example) should be in the state graph before a transformation graph applies, and still be there afterwards.
- The blue dotted edges and nodes (the dotted line last in the example) mean that they should be in the state graph to match this transformation, but should be deleted with the application of the transformation graph.
- The green bold edges and nodes (The bold lined last, val and Object in the example) should be created in the state graph with the application of the transformation graph.
- The red bold dotted edges and nodes (the bold dotted val and Object in the example) mean that if they exist in the state graph, the transformation cannot be applied.

Figure 3 illustrates how the transformation in figure 2 is matched to the state graph. The edges and nodes that are used to match are displayed with bold lines. Note that these are the black and blue dotted edges (and nodes) from the transformation graph.

Figure 3. Example of a GROOVE match on black and blue edges and nodes

After the application of the transformation graph, the state will be changed to the state displayed in figure 4. Note that the bold green edges and nodes will now have appeared in the state graph. The blue dotted parts have been deleted.

Figure 4. GROOVE state graph after application of the transformation graph in figure 2

2.2 Gra2MoL

Gra2MoL stands for Grammar to Model Language and is a rule based language to execute queries on source code, based on the ANTLR grammar that defines this source code. These rules generate a model in the form of an EMF (Eclipse Modeling Framework) model (also known as Ecore model). It also uses a metamodel in the same format, which defines the structure of the resulting model.

Listing 1. Example Gra2MoL rule

```plaintext
rule 'example'
  from grammarElement g
to SCML::ExampleNode
  queries
    sub : //g/#subGrammarElement
    { leaf.exists; }
  mappings
    child = sub;
    codeline = extract g;
end_rule
```

Listing 1 shows an example of a Gra2MoL rule. It exists of 4 sections; from, to, queries and mappings. In the first section of this rule, starting with from, it states that this rule works on the fictional grammar element grammarElement, defining the short name g for easy use in this rule. The second section, starting with to, defines which metamodel element should be used for these grammar elements. In this case, the node ExampleNode of the metamodel SCML is used.

The next part defines zero or more queries on this grammar element. For this, it has three operators for searching through the syntax tree, /, // and ///.

- / searches for direct children of the previous element. In the example, it searches for direct children named subGrammarElement of the element g (grammarElement).
- // searches recursively for the element and returns all elements it found. In the example, it merely defines that it works on the grammarElements for which the rule was called.
- /// searches recursively for an element and returns all direct subnodes (if there are nodes below these subnodes with this name, they are not returned).

This section can also use the # operator, which defines the root node for the identifier of this query. In the example, this means that every subGrammarElement will be
assigned to the identifier sub, not a child or anything else. This is useful for long queries which check for multiple subnodes.

Both the queries section and the from section can use the exists keyword. This can be used to check for the existence of leaves (not nodes). In the example the existence of leaf is checked. In a similar fashion, there is the eq keyword ((leaf.eq("String"))), which checks if the contents of a leaf are equal to a string in the query.

Finally, there is the mappings section. The first mapping maps the query sub to the metamodel reference attribute child. The second line uses the keyword extract to add the source code of this grammar element to the model as a String attribute.

An example of the metamodel that accompanies this rule can be seen in figure 5.

Figure 5. An example of a Gra2MoL metamodel

3. RELATED WORK

There is a verification tool called Bandera which generates models from Java code. It generates multiple forms of models, including Promela, a rich model description language[3]. It does this only for Java, which is why the tool itself it is not suitable for use within this project. The way it generates the models could however be a good starting point for this research. Since the GROOVE graph transformation format has been well defined, it should have concrete hints about how the actual generation of the models could work[4]. Finally, there is a language called Gra2Mol, which is specifically made for the extraction of models from source code[2]. We use this language in this research.

4. PROBLEM STATEMENT

Since making these transformation graphs for an application is a lot of work, it would be useful if these graphs could be generated from the source code of an application. There is currently no tool available which can do this, for any programming language.

A graphic representation of the problem is shown in figure 6. On one side there is the source code, on the other side there is the model (a transformation graph) that is the representation of the code. A tool which uses a generic modelling language is to be created to generate this model.

Figure 6. A global overview of the tool structure

4.1 Research Questions

Following from the problem statement, the main research question is:

- How can a transformation graph be generated from source code?

Some other sub-questions that can be asked:

1. In specific, how can a transformation graph be generated from Java source code?
2. What should a branch (conditional, loop, etc) look like in a transformation?
3. How can the start state be generated from source code?
4. What needs to be done to make sure these transformations can be generated from any language?

5. APPROACH

To define the scope of this research, an overview needs to be made of the whole project within which this research belongs. The design of the tool can be illustrated in four steps; Within this scope, step 4 is the topic of the research in this paper.

1. ANTLR Grammar file parser
   This part reads ANTLR (ANother Tool for Language Recognition) grammar files and provides an interface so that parts of the grammars can be requested.

2. User combination specification
   This part combines grammars with the interface provided in step 1 by using a set of user defined rules. These rules will be evaluated with Prolog.

3. Transformation language
   This part defines a language which can define transformations from ASTs to models representing the source code. This could be Gra2Mol or a new language.

4. Source to graph
   With the language from step 3, some rules should be defined which take source code written in (a subset of) Java and generate transformation graphs from this to be used as input for GROOVE.

To make things clear, the first part of figure 7 shows how steps 1 and 2 are supposed to work. A combination of steps 1 and 2 and the tool mentioned in the problem statement (figure 6) defines the full project. The modelling language mentioned in figure 6 would be defined in step 3. The combination of all steps defines the project, as can be seen in figure 7. Once the final model has been acquired, GROOVE can be used to verify the model.

I will be involved with the implementation of steps 1 and 4 in this approach. Step 1 looks like this:

- Parse grammars; because ANTLR is free and open source software, this can be done by using ANTLRs own source code[1].
- Provide an interface so parts of these grammars can be requested.
Figure 7. The overall process of the program verification project

Step 4 is the main research and it will hopefully answer the main research question, how can a transformation graph be generated from source code? In specific, the following issues need to be addressed:

- Investigate how assignments of basic types (ints, bools, chars) should be represented in a transformation graph.
- Investigate how assignments of Object types (references) should be represented in a transformation graph.
- Investigate how branching should be represented in transformation graphs models.
- GROOVE has some support for typing of object nodes. Investigate if this is useful for generation from code.

These are the most basic subjects that need to be covered before more complex features such as loops can be covered.

To get an idea of what of how specific code structures should be represented, consider code listing 2, a part of a get() method for a LinkedBuffer. This code is represented by the transformation graph in figure 8. The graph follows from these points:

- this.next can not be null, thus it must exist and be a Node of its type, LinkedBuffer.
- this.next.next must exist and be null.
- result is the item that is returned and does not change the state.
- this.next = null means the current reference is deleted and replaced by null. Thus, the LinkedBuffer and all connecting nodes are shown with blue dotted lines.
- return-statements do not affect the state.

Listing 2. Example code snippet

```java
// parts removed
if (this.next != null && this.next.next == null) {
    result = this.next.val;
    this.next = null;
} // parts removed
return result;
```

Figure 8. Transformation graph representing (a part of) a get() method

During the process of creating a graph model like this, with Gra2MoL, it should become clear if Gra2MoL is perfectly suitable to generate models from source code and what drawbacks it has.

Note that the first parts of the project use grammars to define a language and the final step uses these grammars (in the form of an AST (Abstract Syntax Tree) for the source code). However, the rules with which the graphs are generated from the code still depend on the semantics of the grammar. This means that these rules have to be defined by hand for each language that the tool needs to process.

6. RESEARCH

To find an answer to the research questions and a solution to the problem statement, the first thing to be investigated was how we could parse the source code and generate an intermediate representation. For this intermediate model, we use the Gra2MoL output, an EMF model.

Because the model output of Gra2MoL can be defined with the metamodel, as explained in section 2.2, it can be used in theory to generate models that resemble all GROOVE graph models. Because of the versatility of querying the
syntax tree, it should be possible to extract code blocks such as the code in listing 2.

Due to time constraints and the unexpected problems mentioned in section 6.2, the main focus has shifted towards actually being able to generate an abstraction of the syntax tree that more resembles a (transformation) model. The model translation and generation with Gra2MoL has been limited to a number of rules that find assignments, possibly inside an if-block. To keep code in this paper limited, only the relevant Gra2MoL rules have been included. This is detailed in section 6.1.

Figure 9. The process of using Gra2MoL

The overall process of using Gra2MoL and how we can generate a model from source code with it can be seen in figure 9.

6.1 Results

In Java, almost everything is a statement, so the first thing to do is to find all statements. This happens within a block. In code listing 3 we find all statements within a block and assign them in the mappings section as a child of the Block node. The metamodel that goes with this rule can be seen in figure 10. As you can see, blocks can contain any number of SimpleStatements.

Listing 3. Block Gra2MoL rule

```
rule 'methodBlock'
from block b
to SCML::Block
queries
sm : //b///#statement;
mappings
statements = sm;
end_rule
```

Figure 10. Block metamodel

Expressions are all contained in statements, however conditional code such as if-statements are on the same level in the grammar as expressions, they are also statements. To distinguish these from statements that contain expressions, the rule in code listing 4 is used. Statements that have a TOKEN leaf that explicitly marks the statement as an if-statement are translated to a Control Flow Statement, rather than an ImplementationStatement.

Listing 4. Control Flow Gra2MoL rule

```
rule 'controlstatement'
from statement{TOKEN.eq("if")} sif
to SCML::ControlFlowStatement
queries
  subblock : //sif///#block;
mappings
  block = subblock;
end_rule
```

Figure 11. Control Flow metamodel

The metamodel that accompanies this rule is shown in figure 11. It shows that the statement has a Block as a child. A Block can in turn contain statements again.

All other statements are compiled to ImplementationStatements in the rule in listing 5. If they contain expressions (that are assignments) the statements will contain children that are expressions. Figure 12 shows the metamodel for such an ImplementationStatement.

Listing 5. Statement Gra2MoL rule

```
rule 'statement'
from statement s
to SCML::ImplementationStatement
queries
  expl : //s///#expression
  /assignmentOperator ;
mappings
  expression = expl;
  codeline = extract s;
end_rule
```

Figure 12. Statement metamodel

Code listing 6 displays the rule that makes sure that expressions which contain an assignment are displayed in the model. If an expression is in fact an assignment, the left part of this model node (ImplementationExpression) will contain the expression that denotes the variable to which something is assigned. If an expression is not an assignment, any parts that do not match will simply be empty (these nodes will not show up in the resulting model).

Listing 6. Expression Gra2MoL rule

```
rule 'expression'
from expression exp
to SCML::ImplementationExpression
queries
  leftPart : //exp
  /#conditionalExpression ;
middlePart : //exp
  /#assignmentOperator ;
  rightPart : //exp/#expression ;
mappings
  left = leftPart;
```
middle = middlePart;
right = rightPart;
codeline = extract exp;
end_rule

We include the relevant part of the Java grammar here in listing 7 so that it might clarify what an assignment looks like.

Listing 7. Java Assignment grammar

expression : conditionalExpression
            | (assignmentOperator expression)? ;

The metamodel that accompanies this rule is shown in figure 13. It shows that ImplementationExpression has Expression as super type, and that its children (left, middle and right) are again references of the type Expression.

Figure 13. Expression metamodel

With these models, an abstraction is created of the code that contains only blocks, simple statements and expressions. This is a starting point for a GROOVE graph model, at the very least it is already a start state graph.

Figure 14. Example result

With the rules defined here, the resulting model is the tree in figure 14. This tree represents the code mentioned as an example in the approach (section 5, code listing 2) and loosely resembles the GROOVE model, consisting mostly of objects and assignments. The tree describes an if-statement (the Control Flow Statement) which contains a Block with 2 statements in it. These statements contain an expression with three parts, the first describing the variable (identifier), the second the assignment operator, and the third the value (not visible in the image, but these are attributes of the nodes).

If we generate a model in the format GROOVE uses, it looks like the graph in figure 15.

6.2 Drawbacks and challenges

Gra2MoL has given a number of challenges. The query language itself has been documented very well, however the relation to the metamodel it requires it rather unclear.

In general, debug output of what happens during compilation and execution would be a big advantage to the language. Here we list some difficulties that have been encountered during the rule creation process.

- It is unclear when a Gra2MoL rule is called. This is related to both the metamodel as well as the rules. After a while I figured out that a rule is called when the from line matches the root node (denoted with a #) from a rule that is higher in the hierarchy. This root node is assigned to an element in the metamodel. This element is a reference to another type in the metamodel, and has to match the to line in the next rule.

- Complex patterns in the concrete syntax tree are difficult to achieve. This is a drawback in a number of ways.
  - Conditionals in the from line do not have the expected effect. For example, it is possible to say from expression / assignmentOperator, but this will result in a rule on the assignmentOperator node, rather than a rule on all expressions that contain an assignment.
  - On a similar note, it is not possible to go up in the tree, for example to get the expression to which an assignmentOperator belongs.
  - It is also impossible to check for existence of a child node. Gra2MoL has the exists operator (grammarItem(leaf.exists)), but this works on leaves only, not on nodes. Thus it is not possible to use expression (assignmentOperator.exists). One can still use the normal subnode operator (/) in queries, but this is not desirable in the from line because it changes the way the rule works.

6.3 Discussion

The model resulting from the rules in section 6.1 should be compared to the manually created result for this example, as shown in the approach, figure 8. The result tree contains at least enough information that the value of next will be reassigned, becoming null. It does not yet do the necessary checks for the expression inside the if-statement, because the rules for this have not been written yet. These can be written by giving the Control Flow statement an expression child node. Another thing which it does not yet contain is some form of labeling or annotation to easily see or check what an expression defines, in terms of the specific colored lines as used in GROOVE models. These are for example a requirement (black lined), deletion (blue lined) or assignment (green lined). However, this information is available by checking where in the tree the expression occurs (as a subnode of a control flow (if-statement), or standalone as a statement?). With extra Gra2MoL rules, this extra information can be explicitly added to the model.

To get back to the research questions, a GROOVE graph model can be generated by searching for all code which changes something to the state, together with the requirements for these changes, the expressions that define a conditional block. The method described here applies for both the start state as well as the transformation graphs, where the transformation graphs need more annotations (the coloring of the transitions). To make this approach work for other languages, one needs only create extra Gra2MoL rules for that language, using the same metamodel but a
different grammar. This should be an easy task, assuming the language has a similar structure as Java. To look back at the steps of creation mentioned in the approach, we can see that the current state of the rules created treats both object and basic types the same way. We can also see that branching is supported by the Control Flow statements. Since the resulting model is not an actual GROOVE model yet, it does not yet contain specific typing for object nodes. At this point, typing is not needed.

7. CONCLUSIONS
As described in the research and the discussion, we currently have a model that resembles a GROOVE model. It does not display every aspect of a GROOVE graph model yet, but for this we can write extra rules. Gra2MoL is the language that we use for writing these rules, however there are still a number of challenges to overcome when creating rules and models, most importantly the matching of complex syntax tree pattern matching. Any future work on this should lie in improving Gra2MoL in such a way that it can apply the patterns required more easily. The use of the language certainly makes traversing the syntax tree easier, and within the scope of the project is of more use than matching patterns on syntax trees for many different languages manually.

8. REFERENCES