A GrGen.NET solution of the Program Comprehension case for the GraBaTs 2009 Contest

Sebastian Buchwald Edgar Jakumeit Moritz Kroll

May 22, 2009

1 Introduction

The challenge of the Program Comprehension case is to gain specific information out of a given Java program using graph transformation (see [1] for details). After a short description of the GrGen.NET system, we discuss the first part of the challenge and give an introduction into our solution, followed by a presentation of the second part of the challenge and our corresponding solution. Finally, we conclude.

2 What is GrGen.NET?

GrGen.NET is an application domain neutral graph rewrite system developed at the IPD Goos of Universität Karlsruhe (TH), Germany [2]. The feature highlights of GrGen.NET regarding practical relevance are:

Fully Featured Meta Model: GrGen.NET uses attributed and typed multigraphs with multiple inheritance on node/edge types.

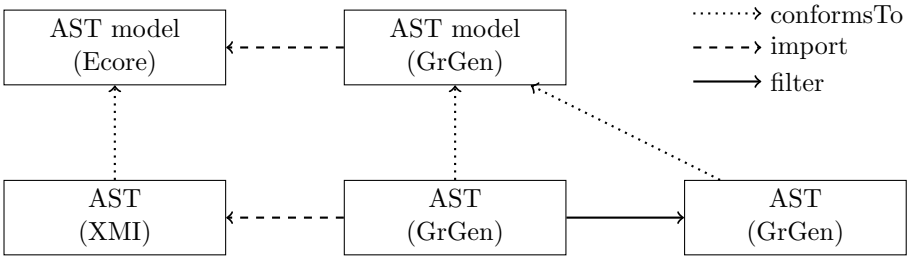
Expressive Rules, Fast Execution: The expressive and easy to learn rule specification language allows straightforward formulation of even complex problems while remaining one of the fastest automatic graph rewrite systems known to us (cf. [3]).

Programmed Rule Application: GrGen.NET has a high-level interface to programmed rule application: Graph Rewrite Sequences (GRS).

Graphical Debugging: GrShell, GrGen.NET's command line shell, offers interactive execution of rules, visualising the current graph and the rewrite process. This way you can see what the graph looks like at a given step of a complex transformation and develop the next step accordingly. Or you can debug your rules.

3 The Filtering Task

The first task is to select all classes that declare a public static method whose return type is the class itself. The goal of the task is to evaluate the scalability of the tool on large graphs, regarding memory consumption and execution speed.



The java programs to be filtered are given as abstract syntax trees of increasing size, each syntax tree following a common model. Both the ast model and the ast itself are given as XMI files which need to be imported into the graph rewriting system first posing a major non-graph rewriting obstacle. As shown in the figure above, the given Ecore model was transformed to a GrGen.NET-specific graph model (.gm-file) by mapping classes with their attributes to corresponding GrGen-classes with attributes, transferring inheritance one-to-one and mapping references to edge classes. Class names are prefixed by the names of the packages they are contained in. The instance graph XMIs following the Ecore XMI meta model are then imported into the system under remapping to the GrGen-model. Now that we have all the information available we can begin with graph rewriting by executing GrGen rewrite rules as they are specified in the task1.grg rule file. The workhorse rule is

```
rule filter(var visID:int, var counter:int):(int) {
  type:TypeDeclaration -:AbstractTypeDeclaration_name-> name:SimpleName;
  method:MethodDeclaration;
  method <-:AbstractTypeDeclaration_bodyDeclarations- type;
  method -:MethodDeclaration_returnType-> returnType:SimpleType;
  returnType -:SimpleType_name-> returnName:SimpleName;
  publicModifier:Modifier <-:BodyDeclaration_modifiers- method;
  staticModifier:Modifier <-:BodyDeclaration_modifiers- method;

  if { !visited(type, visID);
     name.identifier == returnName.identifier;
     publicModifier.public == true;
     staticModifier.static == true;
  }

  /* Rewrite part ... */
}
```

The pattern part is built up of node and edge declarations or references with an intuitive syntax: Nodes are declared by `n:t`, where `n` is an optional node identifier, and `t` its type. An edge `e` with source `x` and target `y` is declared by `x -e:t-> y`, whereas `-->` introduces an anonymous edge of type `Edge`. Nodes and edges are referenced outside their declaration by `n` and `-e->`, respectively. An attribute condition is given within the `if`-clause, constraining the allowed attribute value of some matched elements; here we check that the return type of the method is the class itself and that the method is public and static. Pattern elements and values may be handed in from the outside as parameters.

```
rule filter(var visID:int, var counter:int):(int) {
  /* Pattern part ... */

  modify {
    eval {
      visited(type, visID) = true;
    }
    emit("\t<View:Class␣xmi:id=\"a" + counter +
         "\"␣name=\"" + name.identifier + "\"/>\n");

    return (counter + 1);
  }
}
```

The rewrite part is specified by a `modify`-block nested within the rule. Normally here you would add new graph elements or delete old graph elements, but in this case we only want to mark the found class as already visited. Furthermore we emit the corresponding XMI-tag by using the `emit` statement. Marking the class as visited in the rewrite part and checking for not being visited in the pattern part ensures that each class is emitted only once, even if it possesses several methods which fulfill the filtering criteria.

The benchmark results for the filtering task are given in the following table.

set no.	import time	import size	shell time	shell size	filter time
0	2,142	21	208	34	5
1	3,874	60	785	111	10
2	27,206	445	10,317	924	26
3	60,581	971	30,935	1,962	57
4	64,896	1,049	34,486	2,068	58

Table 1: Results for different input sets; running time in ms, memory usage in MiBytes.

As one can see easily, the time for filtering by application of the graph rewriting rule is negligible and completely dominated by the time needed

for importing the graph. The given values are computed as the arithmetic mean of the middle 3 values out of 5 measurements, on a Core i7 920 with 6 GiBytes of main memory under Windows Vista 64 Bit with MS .NET 64 Bit. Import time is the time needed for importing the graph, import size is the size of the heap after importing the graph. Shell time is the additional time to transform the imported graph as it would show up on API level to a named graph as used by the GrShell of the rapid prototyping environment, shell size is the size of the heap after the named graph was constructed. Filter time is the time for the iterated application of the filtering rule until all matches were found and dumped.

4 The Analysis Task

The goal of the second task is to compute a control flow graph out of the abstract syntax tree and then to compute a program dependence graph out of the control flow graph. Both graphs must be exported as XMI conforming to given graph models. The second task is independent from the first one, but a real world program comprehension task as posed by a software developer working with a program would consist of filtering a given program graph for a criterion currently under interest and then doing a detailed analysis of the filtered result (thus combining both tasks).

The control flow graph is computed by successive transformations of the AST:

1. Compute the infix string representation of all expressions which is necessary to name the CFG nodes.
2. Insert control flow edges into the abstract syntax tree.
3. Retype all AST nodes of relevance to control flow into CFG nodes.
4. Remove remaining AST nodes.

In accordance with the task description, only certain important language constructs (`if`, `while`, ...) are prototypically transformed; taking care of all of the AST nodes would be beyond the scope of this challenge.

Analogously to the first task the XMI representation is created by graph transformation rules with `emit` statements. These rules are controlled by several graph rewrite sequences contained in the GrShell script task2.grs. For example

```
xgrs ((n, id) = CFG_node_by_id(id) && CFG_dump_AbstractNode(n))*
```

iteratively gets the node for an increasing id and dumps it.

Based on the control flow graph, shown in Figure 1, we built the data dependencies and the control dependencies of the program dependence graph.

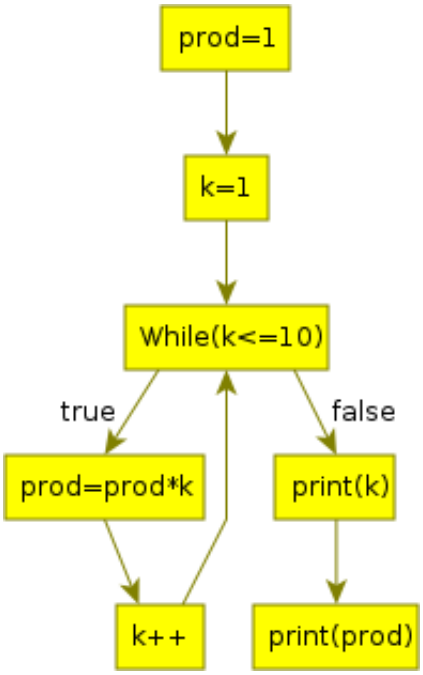


Figure 1: Constructed control flow graph.

Computing the data dependence graph is rather straight-forward and will be shown later on, first we will have a look at computing the control dependence graph which is done in several steps: The first step is to compute post dominance. A statement y post-dominates a statement x iff every path from x to the exit node contains y. This is normally computed by the flow equations:

$$\begin{aligned} PD(exit) &= \{exit\} \\ PD(n) &= \{n\} \cup \left(\bigcap_{p \in succ(n)} PD(p) \right) \quad , n \neq exit \end{aligned}$$

where $PD(n)$ is initialized with all CFG nodes. We implemented this approach with graph transformation: If a node x is contained in the set $PD(n)$ there is an edge $x \rightarrow n$ of type `pdom`. As initialization we use the reflexive transitive closure of the reverse control flow edges which is a less overestimated version as the one described in the equations above (saving us numerous edges thus being more efficient). The intersection is realized stepwise by removing a post dominance edge targeting a node n whose source node x does not postdominate all control flow successors of n and then propagating the removal to transitive edges. The next step is to compute post dominance frontiers for each node n, formally defined as follows:

$$PDF(n) = \{n' \mid \exists x \in succ(n') : n \; pdom \; x \wedge \neg(n \; spdom \; n')\}$$

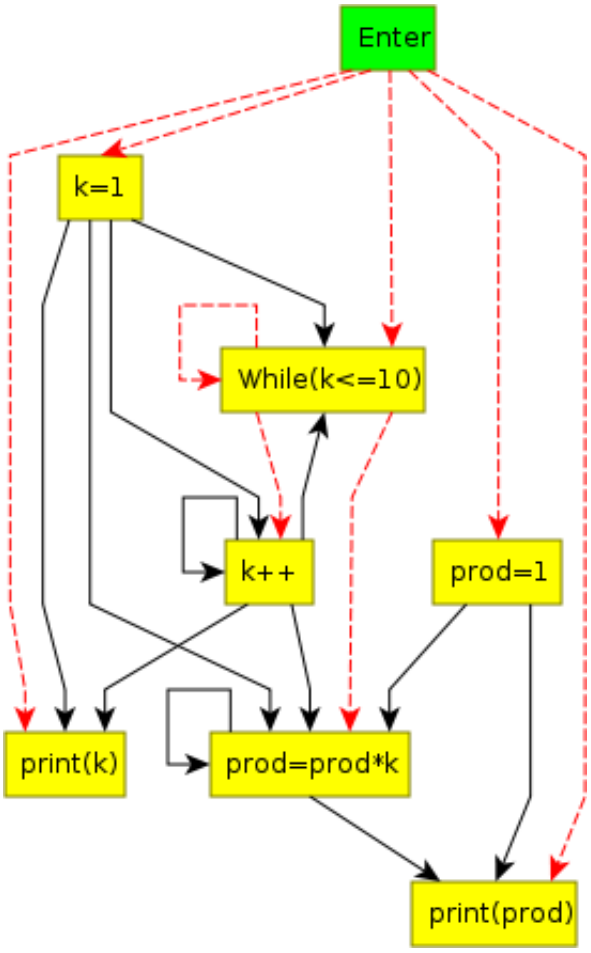


Figure 2: Program dependence graph with control dependence edges (dashed) and data dependence edges.

where spdom denotes strict post dominance, i.e. non-reflexive post dominance. Its graph rewriting realization consists of adding a post dominance frontier edge from a node n to a node n' iff there is a control flow successor node x of n' which is post dominated by n (target of a post-dominance edge from n), but n' is not. Using the fact that n is control dependent on n' iff $n' \in PDF(n)$, we insert the needed control dependence edges into the graph.

Data dependencies were computed the standard way from uses and definitions, i.e. compute use and def for each expression[1], then compute all paths from definitions to uses without another definition of the same variable on it and add a data dependence edge for each such path. Exporting

[1]In order to compute the use and def information for the expressions an adaption of the CFG model was necessary.

was done the same as for the control flow graph. The resulting program dependence graph is shown in Figure 2, whereas control dependencies are denoted by red dashed edges and data dependencies by black edges.

5 Conclusion

In this paper we presented GrGen.NET solutions to task one (filtering) and task two (analysis) of the Program Comprehension challenge, and thus showed the scalability and genericity of GrGen.NET (1+1=5 Points), i.e. that performance is not gained at the expense of reduced expressiveness. Our solution shows, firstly, that the graph rewriting approach is well suited for program comprehension, and secondly, that by now the graph rewriting community has produced tools capable of handling large, real world tasks.

References

[1] Sottet, J.S., Jouaolt, F.: Program comprehension (2009) http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2009reverseengineering.pdf.

[2] Geiß, R.: GrGen. http://www.grgen.net (2008)

[3] Schürr, A., Nagl, M., Zündorf, A., eds.: Applications of Graph Transformation with Industrial Relevance, Proceedings of the Thrird International AGTIVE 2007 Symposium, Schlosshotel am Bergpark Wilhelmshöhe, Kassel, Germany. Volume 5088 of Lecture Notes in Computer Science (LNCS)., Heidelberg, Springer Verlag (2008)