A GrGen.NET solution of the Reengineering Case for the Transformation Tool Contest 2011

Edgar Jakumeit Sebastian Buchwald

May 2, 2011

1 Introduction

The challenge of the Reengineering Case [1] is to extract a state machine model out of the abstract syntax graph of a Java program. The extracted state machine offers a reduced view on the full program graph and thus helps to understand the program regarding the question of interest. We tackle this task employing the general purpose graph rewrite system GrGen.NET (www.grgen.net).

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. Attributes may be typed with one of several basic types, user defined enums, or generic set, map, and array types.

Expressive Rules, Fast Execution: The expressive and easy to learn rule specification language allows straightforward formulation of even complex problems, with an optimized implementation yielding high execution speed at modest memory consumption.

Programmed Rule Application: GrGen.NET supports a high-level rule application control language, Graph Rewrite Sequences (GRS), offering logical, sequential and iterative control plus variables and storages for the communication of processing locations between rules.

Graphical Debugging: GrShell, GrGen.NET's command line shell, offers interactive execution of rules, visualizing together with yComp 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.

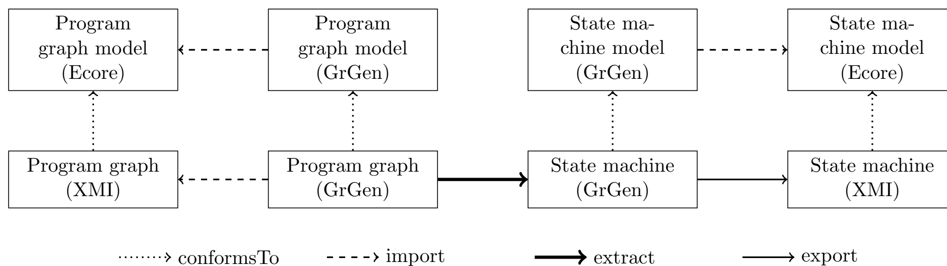

Figure 1: Processing steps of the model extraction. The extraction and the XMI export are written in GrGen.NET languages. Import is handled by a supplied import filter, which generates .gm files as an intermediate step.

3 The Core Assignment

The task of the core assignment is to extract a state machine model out of the abstract syntax graph of a Java program. The task stems from the domain of software reengineering where software engineers need to gain insights into legacy systems, which is a lot easier given a birds eye view on the high level structure (and thus behavior) of the program. The aim of the task is to evaluate the solutions and especially the tools backing them regarding performance and scalability, with a domain leading naturally to large graphs to be considered, and especially to evaluate the solutions and tools regarding the ability to – and conciseness in – carrying out complex, non-local matchings of graph elements, requiring the matching of recursive graph structures. Before the extraction can take place, the Java program graph needs to be imported from an Ecore file describing the source model and an XMI file specifying the graph. Afterwards the resulting state machine has to be exported into an XMI file conforming to a given Ecore file describing the state machine model.

3.1 Importing the Graph

As GrGen.NET is a general purpose graph rewrite system and not a model transformation tool, we do not support importing Ecore metamodels directly. Instead we supply an import filter generating an equivalent GrGen-specific graph model (.gm file) from a given Ecore file by mapping classes to GrGen node classes, their attributes to corresponding GrGen attributes, and their references to GrGen edge classes. Inheritance is transferred one-to-one, and enumerations are mapped to GrGen enums. Class names are prefixed by the names of the packages they are contained in to prevent name clashes; the same holds for references which are prefixed by their node class name. Afterwards the instance graph XMI adhering to the Ecore model thus adhering to the just generated equivalent GrGen graph model is imported

by the filter into the system to serve as the host graph for the following extractions, i.e. transformations. The entire process is shown in Figure 1 above.

3.2 Extraction

The transformation is done in two steps, the first creating the states of the state machine, and the second inserting the transitions in between. Each step consists of the application of one rule (utilizing a subpattern) on all matches found (giving a direct correspondence between coding conventions and rules).

Let us start with a short introduction into the syntax of the basic constructs of the rule language: Rules in GrGen consist of a pattern part specifying the graph pattern to match and a nested rewrite part specifying the changes to be made. 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. Attribute conditions can be given within `if`-clauses.

The rewrite part is specified by a `replace` or `modify` block nested within the rule. With `replace`-mode, graph elements which are referenced within the replace-block are kept, graph elements declared in the replace-block are created, and graph elements declared in the pattern, not referenced in the replace-part are deleted. With `modify`-mode, all graph elements are kept, unless they are specified to be deleted within a `delete()`-statement. Attribute recalculations can be given within an `eval`-statement. These and the language elements we introduce later on are described in more detail in the extensive GrGen.NET user manual [2].

Now let us have a look at the code to create the states:

```
rule createStates
{
  stateClass:Class;
  stateClass -:annotationsAndModifiers-> :Abstract;
  if { stateClass.name == "State"; }

  es:CreateStates(stateClass);

  modify {
    sm:StateMachine;
    es(sm);
  }
}
```

Here and in the following rules the prefixes from name mangling were removed due to space constraints. We search for the abstract class of name `State` as starting point and create the state machine which will receive the states and transitions found. The real work is done in a subpattern `CreateStates`, of which an instance `es` is declared and thus searched from the found `stateClass` on; or better in the rewrite part of this subpattern, which is applied with rule call syntax handing in the just created `StateMachine` node:

```
pattern CreateStates(parentClass:Class) modify(sm:StateMachine)
{
  iterated {
    extendingClass:Class -:extends-> r:NamespaceClassifierReference;
    r -:classifierReferences-> cr:ClassifierReference;
    cr -:target-> parentClass;

    es:CreateStates(extendingClass);

    optional {
      negative {
        extendingClass -:annotationsAndModifiers-> :Abstract;
      }

      modify {
        sm -:states-> s:State;
        s -:link-> extendingClass;
        eval { s.name = extendingClass.name; }
      }
    }

    modify {
      es(sm);
    }
  }

  modify { }
}
```

The subpattern searches all classes directly extending the given parent class handed in as parameter, matching into breadth with the `iterated` construct; the `iterated` matches all instances of its contained pattern which can be found in the host graph. Then the subpattern matches into depth by calling itself recursively with the just matched class as parameter. In the optional case the class is not abstract a state is created within the state machine and a `link` edge is created linking the state with the class. The `optional` matches the contained pattern if it is available in the host graph. The `negative` causes matching of the containing pattern to fail if its pattern can be found in the host graph.

The transitions are inserted with a second rule:

```
rule createTransitions
{
  expressionStatement:ExpressionStatement -:expression-> refTargetClass;
  refTargetClass:IdentifierReference -:target-> targetClass:Class;
  refTargetClass -:next-> callInstance;
  callInstance:MethodCall -:target-> instance:ClassMethod;
  callInstance -:next-> callActivate;
  callActivate:MethodCall -:target-> activate:ClassMethod;
  if { instance.name=="Instance" && activate.name=="activate"; }

  targetClass <-:link- targetState:State;
  def sourceState:State;
  fss:FindSourceState(expressionStatement, yield sourceState);

  sm:StateMachine;

  modify {
    sm -:transitions-> transition:Transition;
    sourceState <-:src- transition -:dst-> targetState;
    sourceState -:out-> transition <-:in- targetState;
    transition -:link-> expressionStatement;
    fss(transition);
  }
}
```

For inserting the transitions into the state machine we search for the `class.Instance().activate()` pattern in the graph, if found we know the target state from the class of the called method and the link between the class and the state we inserted previously. Then we search with the subpattern `FindSourceState` for the source state, which gets `yield`ed into the `def` pattern element `sourceState`. If all of this was found we add a `Transition` in between the source state and the target state, additionally linking it to the `expressionStatement` containing the method call.

```
pattern FindSourceState(containedEntity:Node, def sourceState:State)
                        modify(transition:Transition)
{
  alternative {
    StatementListContainer {
      listContainer:StatementListContainer -:statements-> containedEntity;
      fss:FindSourceState(listContainer, yield sourceState);
      modify {
        transition -:link-> listContainer;
        fss(transition);
      }
    }
    StatementContainer {
      container:StatementContainer -:statement-> containedEntity;
      fss:FindSourceState(container, yield sourceState);
      modify {
        transition -:link-> container;
        fss(transition);
```

```
      }
    }
    StatementSwitch {
      switch:Switch -:cases-> containedEntity;
      fss:FindSourceState(switch, yield sourceState);
      modify {
        transition -:link-> switch;
        fss(transition);
      }
    }
    StatementCondition {
      condition:Condition -:elseStatement-> containedEntity;
      fss:FindSourceState(condition, yield sourceState);
      modify {
        transition -:link-> condition;
        fss(transition);
      }
    }
    StatementTry {
      try:TryBlock -:catcheBlocks-> containedEntity;
      fss:FindSourceState(try, yield sourceState);
      modify {
        transition -:link-> try;
        fss(transition);
      }
    }
    Class {
      cc:Class -:members-> containedEntity;
      ss:State -:link-> cc;
      yield { yield sourceState = ss; }
      modify {
        transition -:link-> cc;
      }
    }
  }
  modify { }
}
```

The subpattern is used to recursively walk outwards from the method call to the class containing the call; passing over the different types of statements and statement containers which might be on the way, until the class is reached yielding it back. The statements passed are all linked to the transition, this will be helpful for the extension tasks.

3.3 Extension tasks

The trigger attribute of the transitions are filled by four rules for the four different ways specified; they get executed one after the other (this way handling the priority), first the non run method, then the switch case, then the catch block and finally the fallback rule. Due to the links from the transitions to all the constructs on the path from the method call to the

containing class this is a simple local pattern search:

```
rule addTriggerNonRunMethodName
{
  transition:Transition -:link-> method:ClassMethod;
  if { method.name != "run"; }

  modify {
    eval { transition.trigger = method.name; }
  }
}

rule addTriggerSwitchCaseEnumValueName
{
  transition:Transition -:link-> case:NormalSwitchCase;
  case -:condition-> caseCondition:IdentifierReference;
  caseCondition -:target-> value:EnumConstant;

  modify {
    eval { transition.trigger = value.name; }
  }
}

rule addTriggerCatchBlockExceptionClassName
{
  transition:Transition -:link-> catchBlock:CatchBlock;
  catchBlock -:parameter-> parameter:OrdinaryParameter;
  parameter -:typeReference-> nspClassRef:NamespaceClassifierReference;
  nspClassRef -:classifierReferences-> classRef:ClassifierReference;
  classRef -:target-> exceptionClass:Class;

  modify {
    eval { transition.trigger = exceptionClass.name; }
  }
}

rule addTriggerOtherwise
{
  transition:Transition;
  if { transition.trigger == null || transition.trigger == ""; }

  modify {
    eval { transition.trigger = "--"; }
  }
}
```

The action attribute of the transitions are filled by two rules for the two different ways specified; first the enum value used in a send method, then the fallback rule. Again this is a simple local pattern search due to the links from the transitions to all the constructs on the path from the method call to the containing class:

```
rule addActionSend
{
  transition:Transition -:link-> block:StatementListContainer;
  block -:statements-> exprStmt:ExpressionStatement;
  exprStmt -:expression-> callMethod:MethodCall;
  callMethod -:target-> method:ClassMethod;
  callMethod -:arguments-> enumClassRef:IdentifierReference;
  enumClassRef -:next-> enumValueRef:IdentifierReference;
  enumValueRef -:target-> enumValue:EnumConstant;
  if { method.name == "send"; }

  modify {
    eval { transition.action = enumValue.name; }
  }
}

rule addActionOtherwise
{
  transition:Transition;
  if { transition.action == null || transition.action == ""; }

  modify {
    eval { transition.action = "--"; }
  }
}
```

A visualization of the resulting state machine is given in Figure 2.

3.4 Exporting the State machine

The XMI export is handled by 5 additional rules given in `export.gri` containing `emit` statements: one for assigning XMI ids to the elements to be exported, which are stored in a map from the nodes to the corresponding ids, two for writing the XMI prefix and suffix, and one each for writing the `States` and writing the `Transitions`, utilizing the previously computed node to id mapping.

4 Rule control, performance, and visualization

The rules are applied from within the graph rewrite script `reengineering.grs` executed by the GrShell, which contains these lines:

```
import primitive_types.ecore java.ecore StateMachine.ecore
         1_small-model.xmi reengineering.grg

xgrs [createStates]
xgrs [createTransitions]
xgrs [addTriggerNonRunMethodName] ;> [addTriggerSwitchCaseEnumValueName] \
       ;> [addTriggerCatchBlockExceptionClassName] ;> [addTriggerOtherwise]
xgrs [addActionSend] ;> [addActionOtherwise]
```

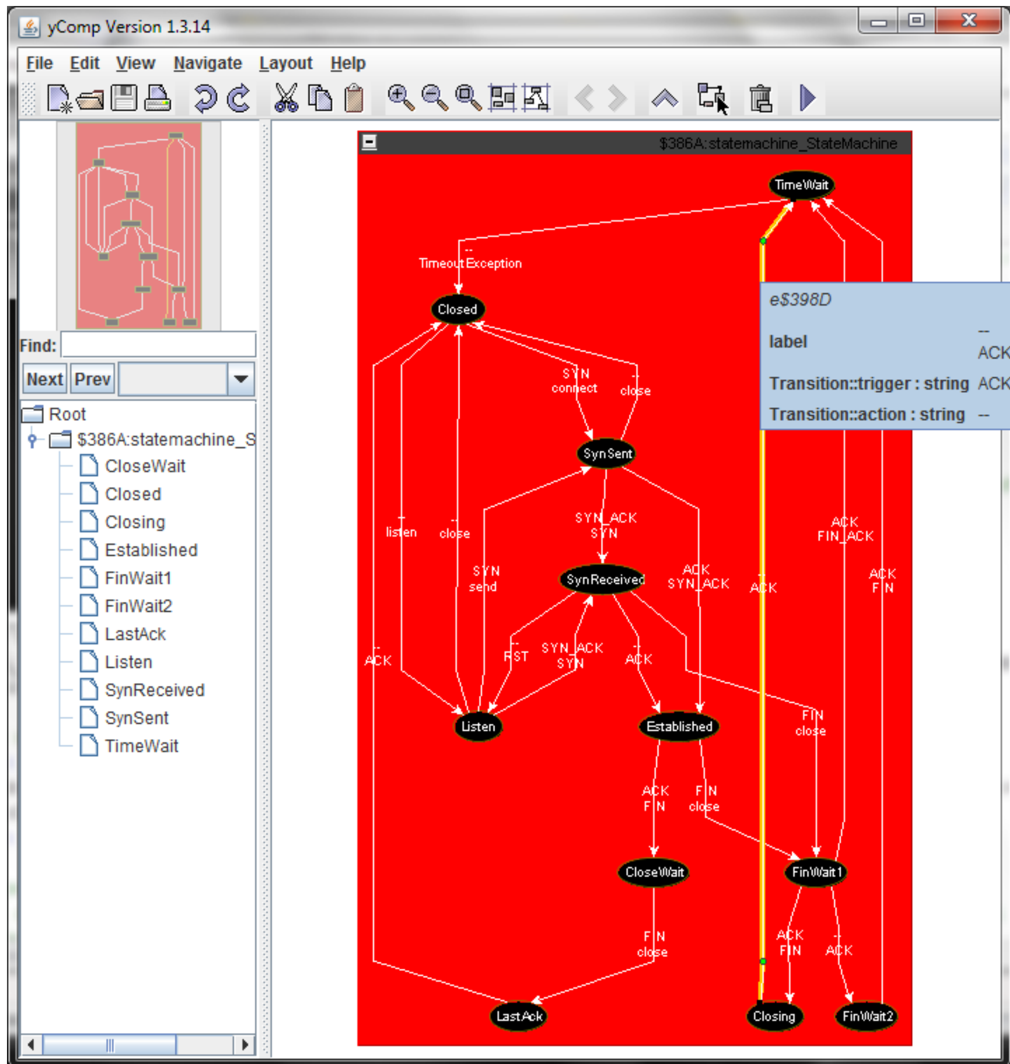

Figure 2: The resulting state machine, with an edge selected and its attributes displayed

The import command imports the XMI input graph complying to the Ecore models, and additionally includes the rules given in the rule file. The `xgrs` keyword starts an extended graph rewrite sequence, which is the rule application control language of GrGen (prepending `debug` before `xgrs` allows you to debug the sequence execution in GrShell). The rules are executed on all the matches found, which is requested by the all-bracketing `[rule]`. The then-left operator `;>` executes the left sequences, then the right sequence, and returns as sequence result the result of the right sequence; the sequence results are uninteresting for this task, in general they are used to control sequence applications.

4.1 Performance

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

set no.	import time	import size	shell time	shell size	extraction time
1	2,855	2.0	31	3.5	130
2	2,917	2.1	32	3.6	140
3	17,878	188.8	4,165	420.9	187
1	1,279	1.3	46	2.3	125
2	1,314	1.3	47	2.4	130
3	28,658	105.3	7,800	277.4	213

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

The given values are computed as the arithmetic mean of the middle 3 values out of 5 measurements, on a Core i7 920 (2.6GHz) with 6 GiBytes of main memory under Windows Vista 64 Bit with MS .NET 64 Bit for the upper part of the table and on a Core 2 Duo U9600 (1.6GHz) with 3 GiBytes of main memory under Windows 7 32 Bit with MS .NET 32 Bit for the lower part. 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 needed 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. Extraction time is the time needed for the application of the extraction rules. Remark: the dominating component of the extraction time is the time needed by the .NET just-in-time compiler producing machine code out of the .NET bytecode.

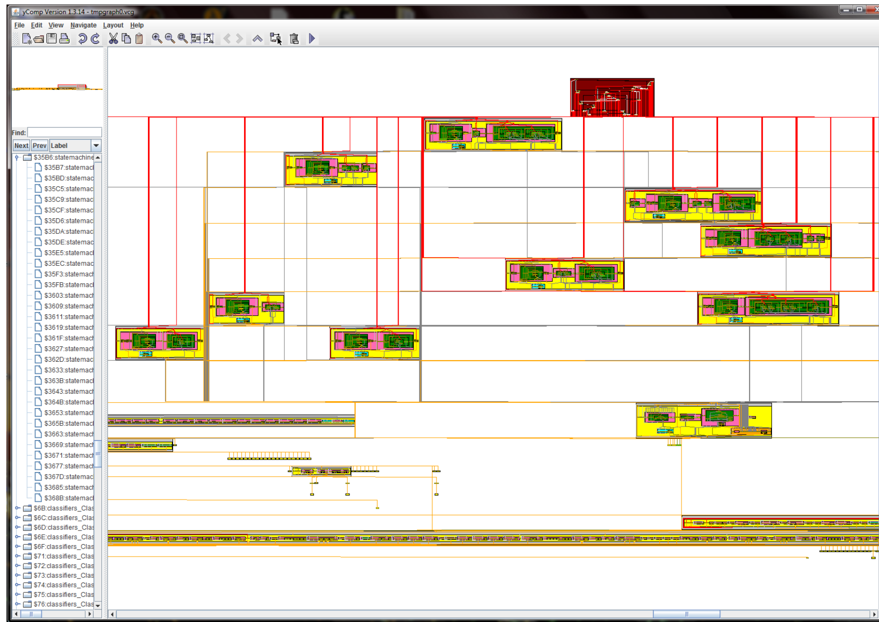

Figure 3: The program graph, the rectangles are the classes, the top one is the state machine

4.2 Visualization

GrGen.NET utilizes the graph viewer yComp as visualization component; the final state machine visualization was already presented with Figure 2. But in addition to the state machine, yComp is able to give a decent visualization of the original program graph, too, as you may see in the Figures 3, 4 and 5 which give a series of images zoomed in, an outstanding help in understanding and debugging.

This is made possible by the high configurability of yComp which gets executed from GrShell. You can use one of several available layout algorithms – with hierarchic, organic and compilergraph being the most useful ones. You can configure for every available node or edge type in which color with what node shape or edge style it should be shown, with what attribute values or fixed text as element labels or tags it is to be displayed, or if it should be shown at all. Furthermore you can configure graph nesting by registering edges at certain nodes to define a containment hierarchy, causing the nodes to become displayed as subgraphs containing the elements to which they are linked by the given edges.

In the following listing, we show an excerpt from our configuration file for customizing the graph layout of the program graph:

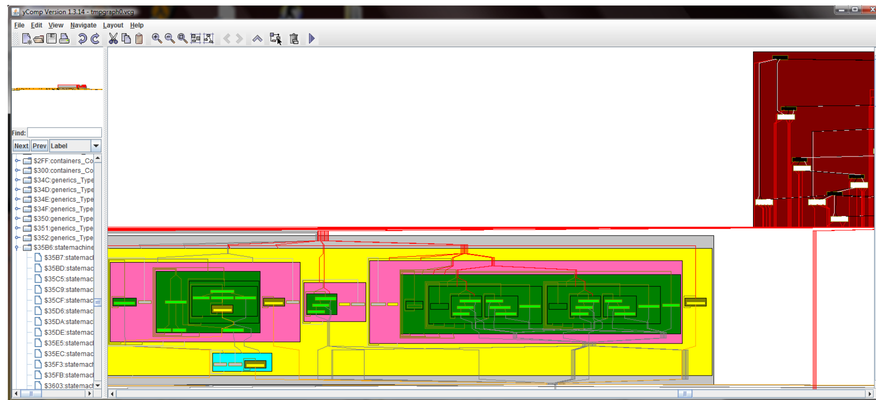

Figure 4: The program graph zoomed with the class SynSent and the state machine

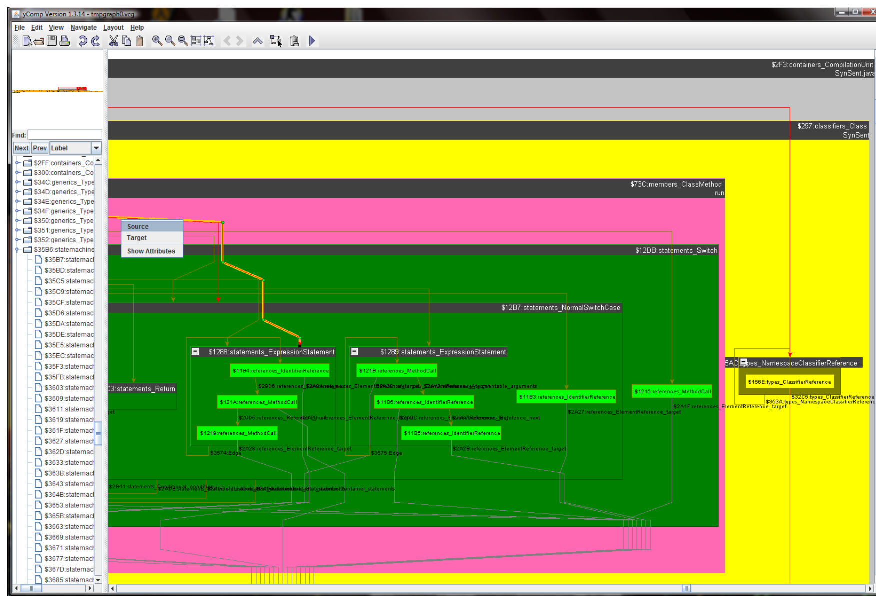

Figure 5: The program graph zoomed further to the method run of the class SynSent

```
debug set layout Hierarchic

dump add node classifiers_Classifier
       group by hidden outgoing members_MemberContainer_members
dump add node members_Method
       group by hidden outgoing members_MemberContainer_members

dump add node classifiers_Annotation exclude

dump set node members_Method color pink
dump set node members_Field color cyan

dump set edge references_ElementReference_target color grey

dump add node classifiers_Class shortinfotag _name
dump add node members_Method shortinfotag _name
```

In addition a helper edge introduction step was added, so that all expression nodes are nested inside their containing statements, not only the outermost ones. A helper step was used in producing the final state machine visualization, too, replacing Transition nodes with real edges.

5 Conclusion

In this paper we presented a GrGen.NET solution to the Reengineering challenge of the Transformation Tool Contest 2011. The abstract Java syntax graph conforming to the `java.ecore` metamodel was imported by a supplied import filter under remapping to the graph concepts supported by GrGen; the extracted state machine was exported by a handful of text emitting graph rewrite rules. A state machine giving a high level overview was extracted out of it using graph rewriting with recursive structures: this ability of matching and rewriting recursive patterns allowed us to give a concise and simple solution to the core task of the Reengineering challenge closely following the specification given, with rules matching kernel patterns and subpattern recursion and iteration to match recursive structures into depth and breadth. During rewriting of the recursive match for transition creation, from the activation call to the containing class outwards, links were inserted from the transition to the elements visited; besides having been a help in debugging they especially allowed to easily solve the extension task with purely local graph rewrite rules. With about 200ms needed for the extraction out of the large graph, performance was not an issue. The goal of the task is to allow program understanding by extracting and displaying a reduced, easily understandable model. In addition to visualizing this simple model we have presented a visualization of the original program graph with our graph viewer yComp, which allows to even work on this level without getting lost in a haystack, what graphs of this size tend to become. This was made possible by color customization and especially graph nesting, grouping

contained nodes into the node containing them, allowing a program reengineer to hierarchically navigate the graph to the point and the elements of interest.

References

[1] Horn, T.: Model Transformations for Program Understanding: A Reengineering Challenge. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/cases/ttc2011_submission_1.zip (2011)

[2] Blomer, J., Geiß, R., Jakumeit, E.: The GrGen.NET User Manual. http://www.grgen.net (2011)