Electronic Communications of the EASST
Volume 1 (2006)

Proceedings of the
Third International Workshop on Graph Based Tools
(GraBaTs 2006)

Automation of Java Code Analysis for Programming Exercises

Carsten Köllmann, Michael Goedicke

11 Pages

Guest Editors: Albert Zündorf, Daniel Varró
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Automation of Java Code Analysis for Programming Exercises

Carsten Köllmann, Michael Goedicke

University of Duisburg-Essen

Abstract:
In this paper we present a tool environment for semi-automatic verification of basic programming exercises. We describe how graph transformation can be used for analysis of code structures and present an example from a current course.

Keywords: graph transformation, static analysis, agg, programming exercises, Java, Tool

1 Introduction

Mapping problems to algorithmic solutions and to program code is a major challenge for first year students. The main problem is the development of an algorithm fitting to the given problem and deriving a corresponding program later on. Experience shows that teaching of abstract structural concepts together with corresponding program structures is an adequate teaching concept.
To internalize this kind of knowledge, numerous accompanying coding exercises are mandatory. Manual correction of these exercises is a protracted work which needs a lot of manpower, especially when the large number of first year students (several hundreds in our case) has to be taken into account. Therefore partial or, if possible, full automation of these corrections would generate a great benefit. Exercises in the field of basic programming are characterized by the fact that a given problem mostly induces only a small number of principal solutions. Furthermore in most cases only a set of similar "standard errors" occurs which can be derived from incorrect usage of the principal solutions the students learned before.
In this paper we describe a graph transformation [EEKR99] based Java code analysis tool for the automatic check of Java applications for "standard errors" specific to a given exercise, so the manual correction later on can focus on more individual problems. The tool supports the full Java 5 syntax and shall provide the basis for more extensive analyses in the future. Its application in the context of exercise testing for a big group of students has been done for collecting experience in a real world scenario. Thus, we can benefit from the results in terms of scalability, the range of possible applications, usage conditions and performance.
The support for structured representation, verification and modification of the Java source code is provided by graph transformation techniques. Graph transformation supports the description of syntax and the formalization of semantic aspects in one coherent formal technique. The representation of source code syntax in a graph structure and its modification by graph transformation has been done in several contexts like program refactoring [ND04] due to its ability of handling structures in a well defined way. Furthermore by using graph transformation it is possible to introduce abstract views on concrete code by e.g. merging elements, or searching for code structures that can occur in arbitrary sequences *without* loosing

the context of the original code. Several existing graph transformation environments give related tool support which allows to actually apply the idea in practice.
Given these advantages we constructed a tool chain based on a general Java to graph structure transformation, structured graph transformation and transformation back to Java and applied this tool chain to the analysis problem sketched above.
In the following we describe the workflow and tools used for the system. We focus on the static Java code analysis by presenting our analysis approach, showing the buildup of an example exercise together with some corresponding verification tests. In addition we present a brief initial evaluation of our approach and justify the main design choices. At the end with give a brief description of our future work directed towards an additional dynamic analysis.

2 Environment

The workflow of our program verification system includes three main components:

- Teacher Component: The teacher creates the coding exercises and corresponding verification tests. For static tests he primary defines the principal solutions of the exercise. Later on graph rules for checking the code for corresponding structures have to be defined by him or a person experienced in creating graph rules. For runtime tests assertions have to be defined which a model checker can check later on. The component stores the rules and assertions for processing students' solutions.
- Student Component: The student uploads the source code of his solution to the server. After performing the checks he gets the information if his code is OK or what kind of standard flaws have been detected. Now he can correct his code and check it again later on.
- Check Component: The Server gets the source code of the students, automatically processes the tests using the tool chain described below and stores the results.

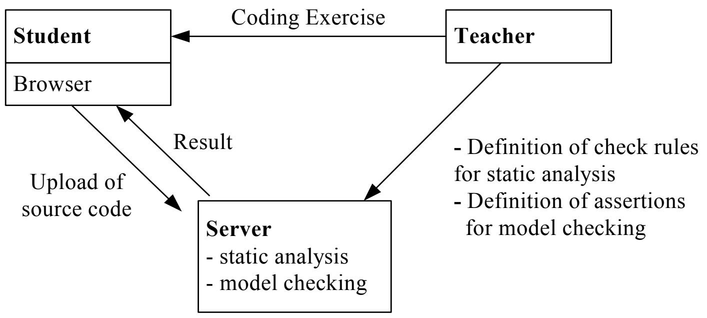

Figure 1: Environment for automatic exercise check

The server component includes automatic verification of the uploaded solution code using rules and assertions defined by the teacher. To realize this fully automated back-end for code verification we combined graph tools for static code verification and for code modifications needed for the model checking [CS01] later on. In the following we describe the workflow realized by the back-end and present a more in-depth view of the tools.

Workflow of the server back-end:

Static Analysis:
After the source code has been uploaded and stored into a database it is transformed into a graph structure with *java2ggx*. Next the graph rules of the specific exercise are applied to the source code graph by using the tool *Control* for choosing the rule(s) to be applied and *AGG* (Attributed Graph Grammar System) [AGG06] for actual processing the selected rule(s). If an error is detected a node is generated including information about the specific problem. After procession of all check-rules the resulting source code graph is parsed for these "error nodes" and their content is stored in the database for generating a report which can be accessed by the submitting student. If no error nodes are found the source code is marked as OK.

Preparing code for model checking:
Because of possible security problems during execution by the model checker *Java PathFinder* (e.g. commands affecting the file system) only the usage of predefined libraries is allowed and has to be checked before with the static analysis techniques mentioned above. Then the assertions a teacher has defined before are inserted into the source code of the student's solution. The assertions are automatically placed in the given code after each assignment of a variable occurring in the assertion. All input operations are replaced with calls to the API of the model checker *Java PathFinder* including the input range that shall be checked. After finishing these modifications the graph structure is retransformed into Java code by *ggx2java*.

Runtime Analysis:
The prepared source code is compiled and the model checker *Java Pathfinder* will be started. If an error has been found the error path is displayed and will be stored into the database.
The server back-end workflow (fig. 2) combines several existing tools and our own tools. The major challenge is the integration of the various tool formats in a way which assures a coherent presentation of the source code along the tool chain in order to generate useful hints related to the submitted original source code in the case of an error. In addition a good performance and scalability has to be accomplished which has been achieved by coupling the different tools as close as possible, e.g. by using directly the API of the graph transformation tool AGG and not its graphical interface.

Thus the server back-end combines the following tools:

- AGG: AGG [Tae00] is a graph transformation tool where graph rules can be applied following the single push-out or double push-out approach [EHK+97]. Graphs in this environment can be attributed by any kind of Java object.

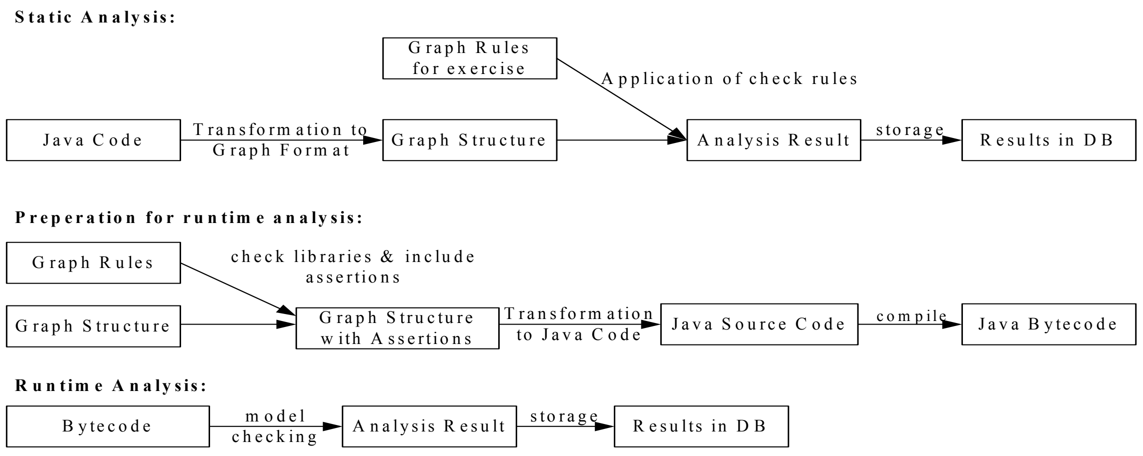

Figure 2: Workflow implemented on server back-end

- Java2ggx: Our tool transforms Java source code into a graph structure and stores it in the graph description format *ggx* used by *AGG*. The code is parsed by using the parser generator *JavaCC* [JCC06] including the Java *TreeBuilder* which provides the Abstract Syntax Tree (AST) of Java. Additional edges are introduced to express implicit dependencies in a source code. Thus is a standard technique which allows easier formalisation of transformation rules. The resulting structure is stored in the *ggx* format.
- ggx2java: Our tool retransforms the *ggx* graph representation into Java source code.
- Control: Our tool uses the AGG API to initiate the execution of a specific sequence of graph rules and therefore the sequence of the source code checks and modifications needed.
- Java PathFinder: Java PathFinder [Path06] is a model checking tool for Java applications, checking assertions and concurrency faults. It executes all potential execution paths of a program for the whole range of possible input values for the application. The range of these values has to be predefined by the user as well as the assertions.

3 Graph based Code Representation and Graph Transformation based Verification

Graph transformations are based on graph grammars that consist of a graph (our Java source code graph representation), and graph rules (our check rules) that perform the transformation. Graph rules consist of a left hand side describing the pattern in a graph to be found, called match, and a right hand side describing the target structure of the transformation.
The graph representation of the Java code is based on its AST. Additional dependencies to the AST are created to simplify the pattern matching for code verification later on. These dependencies include among others explicit edges from object usage to its instantiations and from there to its declaration, edges from method calls to the method declaration and nodes for

explicitly defining the beginning and the end of logical blocks. An example of the source code representation is given in figure 3.
The challenge we are facing with our approach for static source code analysis is how to statically test functional aspects of Java programs that are coded in a very restrictive context. The main idea of our approach is to search for principal structures that should be used for solutions of an exercise and to define those as patterns for the left hand side of a rule. If none of these structures can be found, a possible error is indicated and an error node is created whose content is displayed to the student later on. Searching for these solution patterns by graph rules gets more simple the more dependencies of the code have been made explicit.

The patterns currently used can be applied to five main categories of program structures:

- Intra-method structures: Major structures to be used in a method like different kinds of loops, recursion or condition statements are covered here.
- Inter-method structures: Structures like method calls, parameters, or return statements are checked here.
- Intra-class attributes: The declaration and usage of variables and constants in a class is analysed by patterns covered by this category.
- Inter-class structures: Here the local object instantiations are covered as well as the calls of methods and public attributes from outside.
- Combined structures: More in-depth checks often need a combination of the patterns used above. An example is the verification of the counter variable of a while loop combining an intra-method with an intra-class attributes check.

4 Example of Exercise Creation

In this chapter we present the definition of an example exercise and the creation of its principle solutions and corresponding rules. The exercise has been taken from an existing course teaching basic Java programming. It includes the three steps: a) conceptual exercise formulation, b) definition of principal solutions and corresponding flaws, c) creation of check rules.

Problem statement:
Implement a dynamic FIFO (first in first out) list with methods for insertion of an element, searching for an elements and deletion of an element.

Templates to be used:

```
class Element{
String text;
Element nf;}

class List{
Element head, foot;
void insert (String text){…}
void search (String text){…}
void delete (String tetxt){…}}
```

Possible flaws to check (among others):

1. The implementation does not use the templates.
2. The insert method does not consider the head of the list correctly.
3. The search method includes a loop without a structural correct termination condition.
4. The delete method does not consider the correct handling of the list head.

Definition of check rules and rule flow:

Flaw 1:

- Check if the structure of the given templates appears (separately and in combination).
- Check if the methods are named correctly.

Flaw 2:

- Check if the method *insert* includes a condition containing the sub-term == and a class variable of type Element before or after this expression. The identification of the Element class has to be done by structure check, not by checking the class Name.
- Check if the body of this condition includes assignments from an object of the type *Element* that is dependent on the method parameter, to the class variable detected before.

➔ Possible solution variant:

```
class List{
Element head, foot;
public void insert (String text){
      Element e=new Element;
      e.text=text;
      if(head==null){
head=e;
…
```

Flaw 3:

- Check for loops included in the *search* method and check if there is a connection between the termination condition of the loop and the method parameter.
- Check if on the left or the right of the connected element an object of the *Element* type appears, that points directly or indirectly to the class parameter *text*.

➔ Possible solution variant (see also figure 3):

```
class List{
public void search (String text){
...
while (!e.text.equals(text))
…
```

Flaw 4:

- Check for conditions included in the *delete* method and check if these conditions are connected to class variables
- Check if the corresponding bodies contain a) assignments from these variables and/or b) to these variables.

➔ Possible solution variant:

```
        class List{
        Element head ,foot;
        public void delete (String text){
        if  (e==head){  //e  represents  the  text  containing
element
         head=head.nf; …
```

In the following we show the practical usage of our approach by presenting a description of our implementation for checking flaw 3. Figures 3 and 4 show the main rules used to detect principal solution structures while figure 5 shows the graph of a possible implementation variant. The structure searched for to cover flaw 3 is highlighted here. All figures are from the AGG tool. Some dependency edges from variable usages to their declarations have been hidden in figure 5 for clarity. The graph contains the *Element* class and the *List* class just including the search method.
After the *while* node has been found the corresponding termination condition is analysed. The expression structure is traversed to find a connection to the method parameter *text* and the reference to the *text* definition in the *Element* class.
Figure 3 shows the pattern used to match the reference to the text variable of the *Element* class. The node *mark* is used for traversal through the termination condition (traversal rules are not shown here). We avoided to use the name of the variable in this rule. So even solutions using a different name than *text* for the variable in the *Element* class are found. The bold *text* element of the possible solution expression *while (!e.**text**.equals(text))* is e.g. found here.

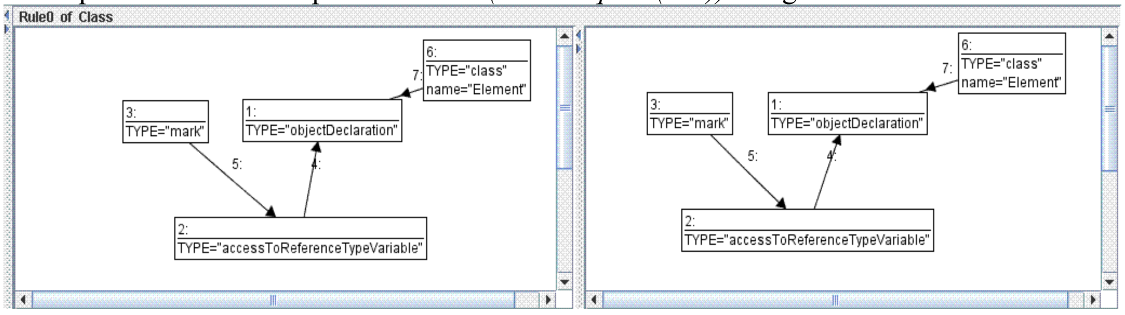

Figure 3: Rule to detect the usage of a variable declared in the *Element* class

The second main rule we used is presented in figure 4. Here the dependency of an element in the termination condition of the while loop on the parameter *text* of the search method is detected. This rules explicitly searches for a method parameter named *text* that has to be used in the condition of the while loop. The bold *text* element of the possible solution expression *while (!e.text.equals(**text**))* is e.g. found by this rule.

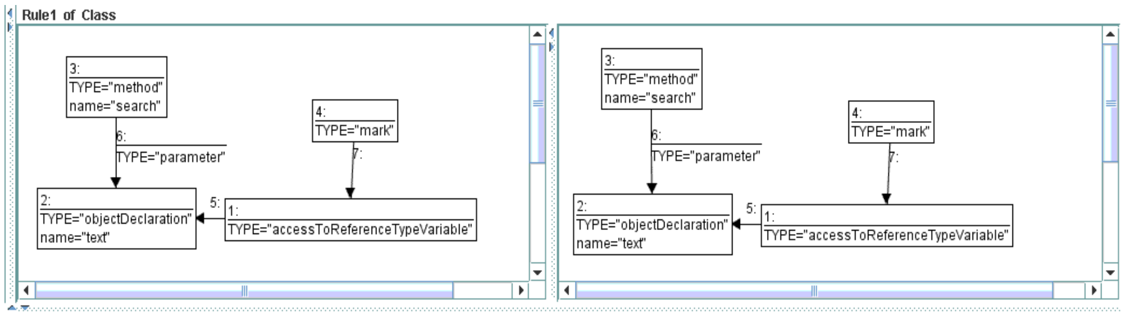
Rule1 of Class
3: TYPE="method" name="search"
6: TYPE="parameter"
4: TYPE="mark"
7:
2: TYPE="objectDeclaration" name="text"
5:
1: TYPE="accessToReferenceTypeVariable"
3: TYPE="method" name="search"
6: TYPE="parameter"
4: TYPE="mark"
7:
2: TYPE="objectDeclaration" name="text"
5:
1: TYPE="accessToReferenceTypeVariable"

Figure 4: Rule to detect the usage of the *text* parameter of the search method

If the two rules described above find a match in the graph of the Java code then the fact is verified, that the termination condition of the while loop contains the correct reference objects. Further rules could now be used to analyse if these are connected with the method call *equals* of the String class. A disadvantage of these rules would be the limited number of possible solutions they cover, because students could e.g. use self written methods for checking the String equality. Of course, more general rules can be formulated which cover more correct solutions variants.
So far we presented the static part of the checking. The dynamic part, using pathfinder, is ongoing work. Here the transformation abilities of AGG are essential for the instrumentation of code with assertions.

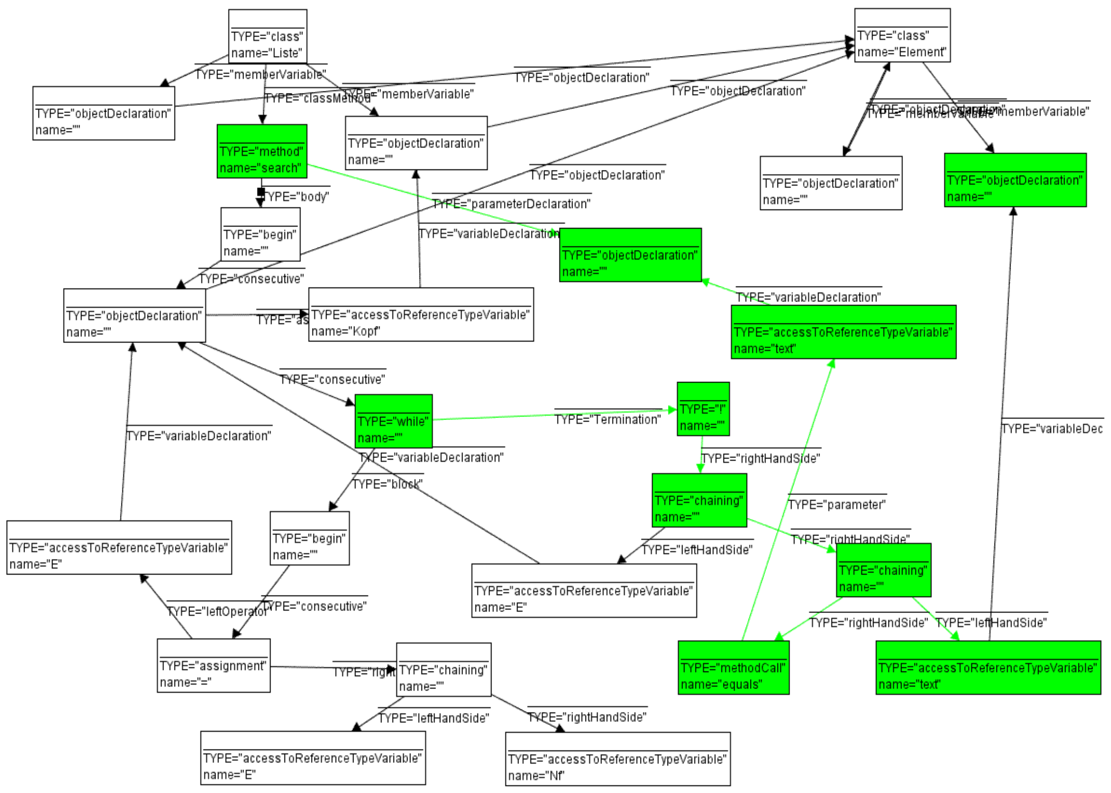
TYPE="class" name="Liste"
TYPE="class" name="Element"
TYPE="memberVariable"
TYPE="classMethod"
TYPE="memberVariable"
TYPE="objectDeclaration"
TYPE="objectDeclaration"
TYPE="objectDeclaration" name=""
TYPE="method" name="search"
TYPE="objectDeclaration" name=""
TYPE="body"
TYPE="objectDeclaration"
TYPE="parameterDeclaration"
TYPE="objectDeclaration" name=""
TYPE="objectDeclaration" name=""
TYPE="begin" name=""
TYPE="variableDeclaration"
TYPE="objectDeclaration" name=""
TYPE="consecutive"
TYPE="variableDeclaration"
TYPE="objectDeclaration" name=""
TYPE="accessToReferenceTypeVariable" name="Kopf"
TYPE="accessToReferenceTypeVariable" name="text"
TYPE="consecutive"
TYPE="while" name=""
TYPE="Termination"
TYPE="!" name=""
TYPE="variableDeclaration"
TYPE="variableDeclaration"
TYPE="rightHandSide"
TYPE="variableDec
TYPE="block"
TYPE="chaining" name=""
TYPE="parameter"
TYPE="accessToReferenceTypeVariable" name="E"
TYPE="begin" name=""
TYPE="leftHandSide"
TYPE="rightHandSide"
TYPE="chaining" name=""
TYPE="accessToReferenceTypeVariable" name="E"
TYPE="leftOperator"
TYPE="consecutive"
TYPE="rightHandSide"
TYPE="leftHandSide"
TYPE="assignment" name="="
TYPE="right
TYPE="chaining" name=""
TYPE="methodCall" name="equals"
TYPE="accessToReferenceTypeVariable" name="text"
TYPE="leftHandSide"
TYPE="rightHandSide"
TYPE="accessToReferenceTypeVariable" name="E"
TYPE="accessToReferenceTypeVariable" name="Nf"

Figure 5: Graph representation of solution code with highlighted nodes for checking flaw 3

5 Evaluation

In the first part of this chapter we desribe an initial experimental evaluation of our approach while in the second part we justify our main design choices.

Experimental evaluation

We evaluated our approach by choosing typical exercises of an introductory programming course. Here we describe two of them. The first one is used in the middle of the semester including the creation of a dynamic list, search for specific elements and deletion of elements in this list. The flaws we cover here have partly been presented in the exercise above. The second example is used at the end of the semester examining the handling of interfaces in Java. Here we checked among other things the valid connection between interfaces and classes and their correct instantiation.

For our evaluation the exercises have been processed by 7 students. Their solutions have been checked by using the tool environment described in chapter 2. Because of the fact that our approach can not process a full semantic check, the solutions have been additionally analysed manually to investigate possible errors at a more detailed level.

The table below includes the number of rules used for checking the whole exercise, the total number of flaws that can be checked by these rules, the number of flaws found during our experiment and the number of false negatives (the students used structures that we have not anticipated during rule creation) found in the student examples. We also counted the number of solutions containing errors that have not been found by our rules because they appeared at a very detailed level.

	Interface Example	**Dynamic List Example**
Total number of rules used for structure check	9	19
Total number of flaws covered	5	10
Flaws found in student solutions	10	6
False negatives found in student solutions	2	2
Errors not found in student solutions	2	1

It can be seen that we needed in both exercises approximately twice as many rules as the number of flaws. The flaws found included wrong class initialisations in the interface example, like *SquareClass q = new GeometricObjectInterface(),* which tries to instatiate an object by using an interface. Further errors in the dynamic list example included a wrong usage of objects, like the statement *head=head.nf* for list traversal during the search for a specific element, which results in the loss of the head element of a dynamic list. Other more simple flaws covered simply missing statements or missing implementation of demanded functionality. The false negatives in the interface example resulted from the usage of the valid array definition *GeometricObjektInterface o [] = {new circle(), new square()}* not covered by our basic array initialisation check in the interface example. The false negatives of the list example resulted from the usage of internal methods not anticipated by us. The errors not found resulted from wrong usage of auxiliary variables.

The result of this preliminary evaluation is encouraging, because of the significant number of flaws found compared to the number of false negatives. However, it can be seen that further abstraction is needed to cover also the false negatives detected here and the handling of auxiliary variables. Of course, this abstraction must not result in false positives. During this evaluation no false positives have been detected.

Justification of the main design choices
Graph transformation is a good choice since it allows addressing structural errors in the context of the student's solutions (see chapter 5). The representation of Java 5 programs using graphs has been chosen to cover a wide range of applications (check and transformation of Java programs). Thus, it may seam that the structure is overly complex. However, the approach was to use the AST plus some additional information to allow a detailed representation. For areas in program checking where a lot of implementation variants have to be covered we use "coarsening" rules, which provide abstractions from the specific graph elements (see chapter 4). The usage of further abstraction techniques is planned to cover information about the software architecture at a more global level e.g. just regarding dependencies between methods and classes.

6 Related Work

In [TRB] a similar approach is presented checking Java code for principal solutions based on the AST and a XML description of the solutions to be checked. The principal solutions are either given as concrete Java code or as an abstract structure. The advantage of our graph transformation based approach is the possibility of using abstract elements together with concrete ones, so it is possible to generate check structures at a more advanced level.
From the point of view of a general E-Learning environment our testing tool helps to ensure the quality of a basic programming course by supporting students in their learning efforts and is therefore part of the quality process [Str06]. Other tools used in this context mostly focus either on static- or runtime analysis whereas we offer a combined solution.
Our approach for identification of principal solution structures in source code can be compared with approaches used for design pattern detection [ACGJ01] and the recognition of "bad smelling code" used for program refactoring [FBB+00]. While these techniques focus on the analysis of arbitrary code, patterns analysed are on a more abstract level like class dependencies or method interactions. Because of the more specific context of our code we can create more in-depth analysis, including more semantic aspects.

7 Remarks and Future Work

In this paper we presented an approach for source code analysis by graph transformation and its applicability in a real world example. We focused on a tool environment combining several existing and self-implemented tools covering the complete Java 5 syntax for verification of first year student exercises.
Currently we implemented the back end tool chain and generated several exercises. The next step is the implementation of the web access and checking the exercises of a whole semester. The verification environment will be used in the basic Java programming course "Programming" in semester 2006/2007 with about 500 students.

Further work will analyse the scalability of this approach detecting the limits of this "exercise specific" process verifying the functional aspects. Furthermore we currently investigate the detection of problematical structures derived from non-functional aspects like performance, and maintainability.

8 References

[ACGJ01] H. Albin-Amiot, P. Cointe, Y.-G. Gueheneuc, and N. Jussien. *Instantiating and detecting design patterns: putting bits and pieces together.* In Proceedings of 16th Annual International Conference of Automated Software Engineering, pages 166-173. IEEE Computer Society Press, November 2001.

[AGG06] The agg web site. *http://tfs.cs.tu-berlin.de/agg/*, June 2006.

[CS01] E. M. Clarke, B. Schlinglo, *Model Checking*, In A. Robinson and A. Voronkov, editors. Handbook of Automated Reasoning, cElsevier Science Publishers B.V., 2001.

[EEKR99] Hartmut Ehrig, Gegor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. Handbook of Graph Grammars and Computing by Graph Transformation, Vol.2: Applications, Languages and Tools. World Scientific, Signapore, 1999.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiron, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: *Single Pushout Approach and Comparison with Double Pushout Approach*. In G. Rozenberg editor. The Handbook of Graph Grammar and Computing by Graph Transformations, Volume I: Foundations.World Scientific. 1996.

[FBB+00] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts*, Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

[JCC06] The JavaCC web site*. https://javacc.dev.java.net/*. June 2006.

[ND04] Niels van Eetvelde and Dirk Jannsens. *Extending graph rewriting for refactoring*. In Proceedings of International Conference of Graph Transformation 2004. Springer. September 2004.

[Path06] The Java PathFinder web site. *http://javapathfinder.sourceforge.net/*. June 2006.

[Str06] C. Stracke, *Process-oriented Quality Management*. In J. Pawlowski, U. Ehlers, editors. European Handbook for Quality and Standardisation in E-Learning, pages 77-91, Springer, 2006.

[TRB04] Nghi Truong, Paul Roe and Peter Bancroft, *Static Analysis of Students' Java Programs*. In Proc. Sixth Australasian Computing Education Conference (ACE2004), Dunedin, New Zealand. CRPIT, 30. Lister, R. and Young, A. L., Eds., ACS. 317-325. 2004.

[Tae00] Gabriele Taenzer. *AGG: A tool environment for algebraic graph transformation*. In M.Nagel, A.Schürr, and M. Münch, editors. Application of Graph Transformation with Industrial Relevance: International Workshop, AGTIVE'99, Kerlkerade, The Netherlands, volume 1779, pages 481-488. Springer, Heidelberg, 2000.