A Review of Static Analysis Approaches for Programming Exercises

Michael Striewe and Michael Goedicke

University of Duisburg-Essen, Germany
{michael.striewe,michael.goedicke}uni-due.de

Abstract. Static source code analysis is a common feature in automated grading and tutoring systems for programming exercises. Different approaches and tools are used in this area, each with individual benefits and drawbacks, which have direct influence on the quality of assessment feedback. In this paper, different principal approaches and different tools for static analysis are presented, evaluated and compared regarding their usefulness in learning scenarios. The goal is to draw a connection between the technical outcomes of source code analysis and the didactical benefits that can be gained from it for programming education and feedback generation.

1 Introduction

Automated grading and assessment tools for programming exercises are in use in many ways in higher education. Surveys from 2005 [3] and 2010 [16] list a significant amount of different systems and numbers have grown since then. One of the most common features of systems for automated grading of programming exercises is static analysis of source code. The range includes checks for syntactical correctness of source code up to checks for structural similarities between a student's solution and a sample solution [32]. Different approaches are used and different tools and techniques are integrated into these systems. For each decision for a tool or technique individual positive and negative effects on the quality of feedback given by the system can be assumed. However, reviews and comparisons of program analysis tools usually focus on bug finding quality in the context of industrial applications by running some kinds of benchmark contests (e.g. [24]) or analyzing case studies (e.g. [1]). Thus it is the goal of this paper to compare and evaluate different principal approaches to static code analysis specifically in the context of automated grading and assessment. Special attention is paid to the connections between technical outcomes of source code analysis and the didactical benefits that can be gained from it for programming education and feedback generation.

This paper focuses on techniques applicable in automated grading and assessment systems that are running as a server application, allowing on-line submission of exercise solutions. We are not concerned with analysis and feedback mechanisms integrated into special IDEs as learning environments. To ensure a

M. Kalz and E. Ras (Eds.): CAA 2014, CCIS 439, pp. 100–113, 2014.

reasonable limited scope, this paper also focuses on approaches and tools useful in the context of object-oriented programming with Java. Results may be partially valid for other object-oriented programming languages than Java. Similarly, some results may be partially valid for static analysis for other programming paradigms.

Static analysis capabilities of tools for automated grading and assessment have also been reviewed in the context of structural similarity analysis [22]. This type of analysis intends to give hints on the systematic extension of incomplete solutions as also considered in this paper. Another large branch of static analysis in learning scenarios is the use of metrics (e.g. [20]). Research and application in this area is more focused on an overall quality measure for solutions than in detailed feedback for single mistakes and will not be considered in this paper.

This contribution is organized as follows: Section 2 gives an overview on the special requirements of static analysis of source code in the context of automated grading and assessment. It also gives an overview on prominent systems for automated grading and tutoring. Section 3 discusses differences between approaches, such as differences between analysis of source code and byte code. These comparisons are made as independently from actual tools as possible. Section 4 discusses features of several tools which are known to be used in current automated grading and tutoring system. Section 5 concludes the paper.

2 Static Analysis in Automated Grading and Tutoring

The goal of automated grading and tutoring tools in learning scenarios is twofold: First, automated tutoring is intended to enable students to develop correct solutions for exercises without intensive assistance by a human teacher. Thus it focuses on giving useful hints on incorrect and incomplete solutions that go beyond plain messages like "error in line X". Second, automated grading is intended to assist teachers in the tedious task of grading large numbers of assignments, especially if formative assessments are conducted several times in a course. In this scenario it focuses on giving adequate marks for solutions, which especially includes distinctions between major and minor errors. The common ground for both scenarios is to generate meaningful feedback automatically, based on a thorough analysis of source code submitted by students.

```
int x,y,z = 0;
// << some code here >>
if (x + y < y + z);
{
  x = y - z;
}
```

Listing 1.1. A piece of Java source code which is syntactically correct, but contains a completely useless `if`-statement

The most basic way of giving feedback to a solution of a programming exercise are reports on syntactical errors as generated by a compiler. For many students, writing syntactically correct code is the first obstacle in learning programming [9] and thus compiler messages are the first automated feedback they see. As this type of feedback can be generated locally on the student's own computer it is of minor importance for on-line submission systems. Anyway, compiler messages as feedback on programming errors are not specific to learning scenarios. Instead, more specific requirements for automated feedback can be derived from learning scenarios:

- Static analysis can check for source code which is syntactically correct but shows misunderstood concepts. A typical example for Java is shown in Listing 1.1. Even an experienced teacher may need some time to realize that this `if`-statement is useless because of the extra semicolon at the end of its line. Mistakes like this can be detected by static analysis and reported in conjunction with a short explanation of the related concepts. The same applies for violated coding conventions. Similar to compiler messages, detecting this kind of mistakes is not necessarily specific to learning scenarios, as these mistakes can in general also be made by experienced programmers. However, we can state as a requirement, that static analysis in learning scenarios needs to check for more than syntactical errors. As a second requirement we can also state that static analysis in learning scenarios must be able to give feedback to parts of the program that have no relevant functionality.
- Static analysis can check for source code which is correct in general terms, but not allowed in the context of a certain exercise or execution environment. For example, an exercise may ask students to implement a linked list on their own. Obviously, the use of `java.util.LinkedList` should not be allowed in this case. In contrast to the requirement discussed above, this is no general coding convention, but specific to a particular exercise. Other exercises may allow to use this existing implementation. Thus static analysis in learning scenarios needs to be easily configurable for each specific exercise.
- Similar to the requirement discussed above, there may be code structures that are required in any correct solution of an exercise. For example, an exercise may ask students to solve a problem by implementing a recursive algorithm. In this case, any solution that does not involve recursion is wrong in terms of the task description, even if the running program produces the correct output. Hence static analysis in learning scenarios must be able to report not only the presence of undesired code structures, but also the absence of desired code structures.
- In tutoring scenarios students may expect to be not only informed about the existence of a mistake, but to get hints on how to correct this mistake and improve their solution. This is especially true for solutions that are correct in syntax and functionality, but do not completely fulfill the requirements for the given exercise. In these cases, students may expect to get a hint on the next step to be taken. Thus the most sophisticated requirement for static analysis is to give feedback on how to systematically extend an incomplete piece of source code to reach a given goal.

Note that there is at least one more requirement in automated grading and assessment systems which involves source code analysis: Checks for plagiarism. We leave this (and similar requirements) out of the scope of this paper, since the required analysis is of different nature than the others discussed in this paper. Checks for plagiarism in general include comparisons between many solutions created by students instead of analysis of a single solution or a comparison between one student's solution and a sample solution. For a study on plagiarism detection tools in automated grading systems refer e.g. to [13].

Not only requirements in the context of automated grading and tutoring can be characterized, but also typical properties of solutions submitted by students. In most cases, automated tools are used in the context of introductory courses, where large numbers of solutions have to be graded. Exercises in these courses are of moderate complexity, so solutions do not consist of more than a few Java classes and a few methods in each of these classes. Enhanced concepts like Aspect Oriented Programming or reflection are typically not among the topics of these courses, so there is no need to care about these in static analysis. Solutions are often created based on code templates or at least prescribed method signatures, so assumptions about existing names for methods and perhaps variables can be used as an entry point for static analysis. As already mentioned above, checking the existence of such prescribed structures is an explicit requirement in grading and tutoring.

Table 1. Static code analysis capabilities of some automated grading and tutoring systems

Name	Source Code Analysis	Byte Code Analysis
ASB	yes (CheckStyle)	yes (FindBugs)
CourseMarker	yes	no
Duesie	yes (PMD)	no
EASy	no	yes (FindBugs)
ELP	yes	no
JACK	yes	no
Marmoset	no	yes (FindBugs)
Praktomat	yes (CheckStyle)	no
Web-CAT	yes (CheckStyle/PMD)	yes

From the literature the following automated grading and tutoring systems for Java could be reported (in alphabetical order): ASAP [10], ASB [21], BOSS [17], CourseMarker [14], Duesie [15], EASy [12], eduComponents [4], ELP [31], GATE [28], JACK [29], Marmoset [27], Mooshak [19], Online Judge [6], Praktomat [33], Web-CAT [26], xLx [25]. Table 1 gives a more detailed overview on those tools that involve more static code analysis capabilities than plain compiler checks. The use of other external tools than CheckStyle [7], FindBugs [11], and PMD [23] could not be found in the literature. All three tools are open source and non-commercial projects. CourseMarker and ELP employ software metrics for static analysis. ELP uses a XML representation of the abstract syntax tree for

this purpose and offers also comparison of syntax trees for students' solutions and sample solutions [32]. JACK uses a graph transformation engine [18] and the graph query language GReQL [5] for analysis of abstract syntax graphs, which are abstract syntax trees enriched by additional elements. We will elaborate more on this later on.

All systems listed above understand static code analysis in automated grading primarily as applying rule based checks. All tools named above do also handle code analysis as rule based or query based inspection, respectively. Consequently, Section 3 and Section 4 of this paper focus on rule based checks as well.

3 Comparing Approaches

This section compares technical approaches used in the tools and systems identified above. Comparison is focused on the general benefits and drawbacks of a specific technique, ignoring limitations or extensions raising from a specific implementation of that technique.

3.1 Source Code vs. Byte Code Analysis

As already suggested by the layout of Table 1 it is important to know whether static code analysis is carried out on source code or byte code. For programming languages other than Java, which are not considered in this paper, byte code may be replaced by machine code. While source code is directly written by students, byte code is generated from the source code by a compiler. Thus the first question to answer is whether byte code can be generated in any case. Since we restricted ourselves to on-line submission systems and assumed students to be able to compile source code on their own, we can also assume that submitted solutions do not contain compiler errors. Thus byte code of a complete solution can be generated and byte code analysis tools have no disadvantage in comparison to source code analysis tools regarding this aspect.

Regarding checking capabilities beyond syntactical checks both source code and byte code analysis are able to report more than syntactical errors. For example, inheritance structures, number of method parameters or types of fields are visible both in source code and in byte code.

Regarding feedback on irrelevant code statements it is important to know that a compiler may be able to remove unnecessary statements for code optimization. While this is beneficial for several reasons in productive environments, it may be a drawback in learning scenarios: Static analysis on byte code is not necessarily able to report unnecessary statements, if these are removed by the compiler. If the compiler gives a notice about removed statements, these messages can of course be used as feedback messages to students. Source code analysis can give feedback on unnecessary statements without general limitations.

Regarding configurability with respect to individual exercises it can be observed that exercise specific hooks like names for classes, methods, or fields are available both in source code and in byte code. Technically there is no major

difference in analyzing e.g. the parse tree of source code or its related byte code. So if a flexible and configurable way of defining checks exist, it can be used for both formats.

Regarding feedback on missing statements the desired granularity has to be taken into account. For example, any kind of loop statement is represented by `goto`-statements in byte code. If a task description requires to use a loop, but there is no `goto`-statement in the byte code, this can be reported as a mistake. However, if the task description requires to use a specific type of loop, it cannot reliably be derived from an existing `goto`-statement, whether this specific type of loop has been used. Although all loop constructs in Java result in typical byte code patterns, analysis of these patterns is not trivial in all cases. In source code analysis, this problem does not exist, since every statement can be recognized from the source code directly.

Regarding hints on systematic extension of incomplete solutions the same concerns as above have to be applied. By comparison of a student's solution and a sample solution a missing loop can be determined both in source code or in byte code. In this case the system can suggest to think about loops. However, if both solutions contain a loop, only source code analysis is able to give more specific hints on completing a certain type of loop, e.g. by detecting a missing termination condition in a `for`-statement.

In summary, byte code analysis does not fulfill all requirements for learning scenarios, while source code analysis seems to do so with respect to all aspects.

3.2 Trees vs. Graphs

As mentioned towards the end of Section 2, there are approaches using an abstract syntax tree, while other approaches use an abstract syntax graph. An abstract syntax graph is basically an abstract syntax tree, which is enriched by additional arcs, e.g. for connecting method call nodes to the respective method declaration or accesses to fields to the respective field declaration [30]. See Figure 1 for an illustrating example. Solid arcs belong to the abstract syntax tree, while dashed arcs extend this tree to an abstract syntax graph. The information used for insertion of this arcs is computed in a post-processing step after parsing by resolving names and scopes. Hence it has to be noticed that the difference between graphs and trees is mainly a difference of data formats. In fact, syntax graphs are generated from syntax trees, so any information available in the graph is also available in the tree. However, it can be considered to make a difference whether this information is available explicitly or implicitly.

The capabilities of checking for more than syntactical errors are not affected by the choice of data format. The same is true for capabilities in reporting missing elements of a solution, because in both cases basically the same structures have to be searched. Configurability with respect to individual exercises is also not affected by the choice of data format.

Irrelevant pieces of code can possibly be found more easily in syntax graphs, e.g. unused methods can be detected by searching method declaration nodes without incoming arcs from respective method call nodes. Hints on systematic

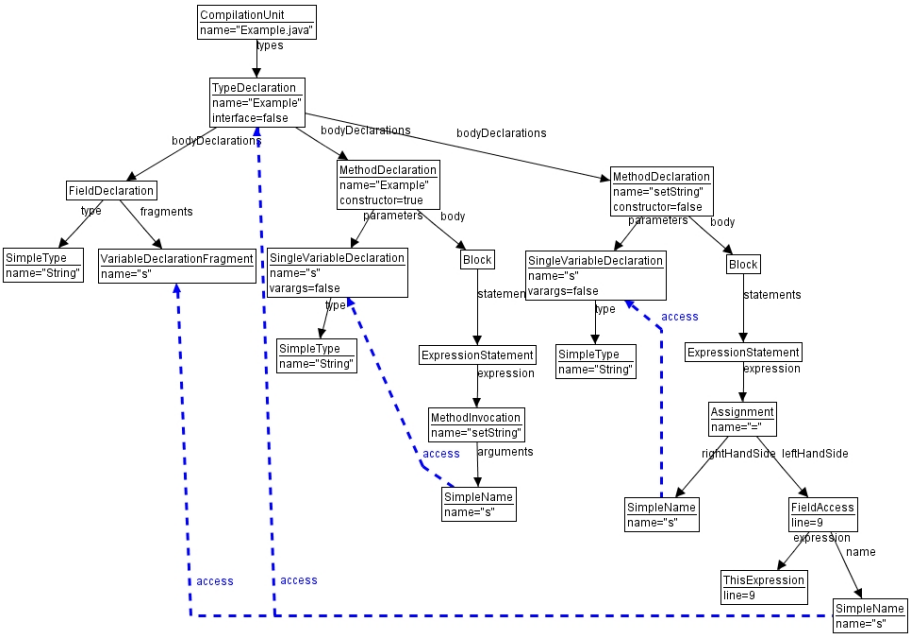

Fig. 1. Example showing an abstract syntax graph for a simple Java class with a constructor and a method. The solid arcs form the underlying abstract syntax tree.

extension of incomplete solutions can have benefits from this fact, because this way hints on missing connections between parts of a solution can easily be given. So generating abstract syntax graphs from abstract syntax trees before starting an analysis seems to be a valuable preprocessing step, which makes some operations easier. However, it does not add functional benefits in the learning scenario. Another aspect is discussed in Section 4.2 later in this paper.

3.3 Single File vs. Multi File Analysis

From the tools discussed in this paper, CheckStyle limits itself to checking only single source files, while all other tools allow to analyze multiple files. Since automated grading is often used in courses with several hundred students, analysis time is a limited resource. Time can possibly be saved by performing analysis in parallel, which is easier if only single files have to be handled. Thus it is a reasonable question whether multi file analysis is necessary because of other requirements of the learning scenario.

The goal of checking for more than syntactical errors is not affected by this question, since other mistakes can also be found in single files. In fact, many solutions of simple programming exercises do not consist of more than one source file at all and static program analysis is not blocked this way.

Finding irrelevant code statements is much harder when single file analysis is applied. For example a method may appear unused in a single file because

it is not called by the class defined in this file, but at the same time it can be called from another class defined in a separate file. To handle this issue, storing results from each file analysis and reviewing this intermediate results would be necessary. The same applies for the search for missing elements, if the task description does not state a specific class where the element has to be located. If no intermediate results are stored, some properties of a solution cannot be assessed. Consequently, configurability for individual exercises can considered to be decreased with single file analysis in this case.

Giving hints on systematic extensions of an incomplete solution based on the comparison to a sample solution is not affected by single or multi file analysis. The total number of features compared may be reduced because of the reasons given above, but each feature found in a single file of the sample solution and missing in the student's solution can be used for directing feedback.

4 Comparing Tool Features

In addition to general benefits and drawbacks of analysis approaches, tool specific issues have to be taken into account when integrating static checks into automated grading and tutoring systems. This integration covers both technical and organizational aspects: Technically, solution data has to be passed from the systems to analysis tools and analysis results have to be passed back to the systems. Regarding organization, tools have to be configured for individual exercises and results have to be interpreted with respect to marking schemes. All these aspects are investigated in this section based on the five tools named already above (Section 2): CheckStyle, PMD, FindBugs, GReQL, and graph transformations. For the latter, graph transformation rules written in AGG [2] are taken into account. Other tools for graph transformations exist, but to the best of the authors knowledge they are not used for static code analysis in automated grading and tutoring systems.

From these five tools, the first three are dedicated code analysis tools and do thus provide features specific for this domain. GReQL and graph transformations are general approaches for handling graphs, which can be used for checking syntax graphs. However, they do not provide any features specific to static source analysis natively and hence they require additional programming effort before they can be used in automated grading and tutoring systems.

Quality of analysis results in terms of false positives or false negatives is not considered in this paper, because they do not only depend on general capabilities of tools and approaches, but on the quality of individual checking rules. Writing precise rules surely requires a good and powerful tool, but also experience and domain knowledge. Thus it is beyond the scope of this paper dealing with approaches and tools from a technical point of view.

4.1 Tool Integration

In general, two different ways exist to integrate an external tool into an existing system: Integrating the external tool as a library and using its API or assuming

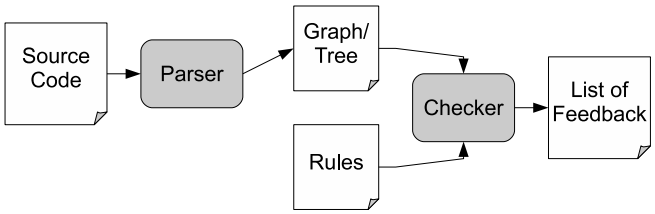

Fig. 2. Analysis process in case of tool integration via an API

it as an existing separate installation and starting it from the command line as a separate process. Further integration with respect to LMS is not considered here, as this has to be done on the level of automated assessment tools as a whole and not on the level of specific checking facilities.

CheckStyle, PMD, and FindBugs can be used via the command line as well as via an API. GReQL as a query language can be executed via a library named JGraLab, thus only API integration is possible. AGG does not offer possibilities for being used via the command line, so it has to be integrated as library, too. Figure 2 illustrates the general process of analysis in cases where the analysis tool is connected via an API.

Regarding feedback quality for static analysis, these differences do not matter. Once the integration is done, no further technical changes have to be applied when the system is used. Since all tools offer API integration, no relevant limitations regarding command line options or possible run time environments for installation of these tools have to be obeyed.

Another aspect of integration is the aspect of semantics of checking rules and results of checks. In CheckStyle, PMD and FindBugs it is clearly defined how rules are applied and which results are returned when a rule matches or is violated. Operations for executing specific checks and obtaining the results are offered directly via the API or via command line options and result files of a specified format, respectively. Different to that, the integration of GReQL and AGG into automated grading and tutoring tools is completely left to the developer. The APIs just offer general methods for executing queries or transformation rules, respectively. If a single check in terms of static program analysis is broken down into several queries or transformation rules, the correct execution of the checking process has to be handled by the developer of the tool integration. The same applies for the interpretation of analysis results. In particular, GReQL and AGG can be integrated in the following way:

– **AGG:** In AGG, rules applied during analysis can be realized by graph transformation rules which introduce additional nodes (e.g. error markers) into the abstract syntax graph. Since these markers can be reused and removed by other rules, this allows for chaining AGG rules to represent more complex analysis rules. If all rules have been applied, the remaining error markers are collected and a list of feedback can be assembled from the error messages contained in each marker.

– **GReQL:** In GReQL, queries on graph structures can be written (somewhat similar to SQL) that report tuples of nodes that match the query. Hence analysis rules can be expressed through a graph query and an expected result, which may be an empty set if the query looks for undesired code structures. If the actual result of the query does not match the expected result, a feedback message is added to the list of feedback, which is returned at the end of the process after all queries have been executed.

On the one hand, this requires much more effort in integration than with dedicated analysis tools. On the other hand, this allows for more freedom in defining complex input and interpreting results.

In summary, these results are not surprising. The more general a tool is, the more effort is necessary to perform specialized tasks. However, since learning scenarios may require very specialized and even exercise specific checks which are not among the standard checks offered by dedicated program analysis tools, the higher effort in tool integration can save effort in productive use.

4.2 Writing Checks and Feedback Rules

One of the requirements as listed in Section 2 is configurability for individual exercises. Thus it is an important question how easy and flexible checking rules can be written for specific tools. Since exercise specific feedback can only be given if exercise specific checks are created, this is a core criterion. As discussed at the beginning of this paper, this focus on feedback is a key difference between industrial use of static analysis tools and use of these tools in e-assessment.

In CheckStyle and PMD, checking rules are implemented using the visitor pattern which traverses the syntax tree. Both tools come with a large predefined set of standard checks, which can be switched on and off as needed. Writing own checks is possible by defining own operations for the visitor and integrating the new implementation to the existing installation via a configuration file. As an alternative, PMD also offers the possibility to define checks as XPath queries on the syntax tree. These additional queries are also integrated by adding them to the configuration. See Listing 1.2 for an example of an XPath query looking for a broken `if`-statement as shown in Listing 1.1. FindBugs offers also a predefined set of checks, but no simple facility to add own checks by implementing new operations. Customizing FindBugs for individual exercises is thus not possible.

As more general approaches, GReQL and AGG offer native support for defining own rules and queries. In fact, GReQL as a query language does not offer anything else than executing queries on graphs in a specified language and reporting results as tuples of nodes as described above. See Listing 1.3 for an example for a GReQL query looking for a broken `if`-statement as shown in Listing 1.1. The rule looks somewhat more complex than the one for PMD, but this is no general observation. In fact, some of the rules built-in to PMD are implemented directly in Java, because an XPath query for them would be too complex [8]. GReQL allows to implement additional functions that can be used in queries to realize complex checks, which allows to shorten queries as well.

```
//IfStatement[@Else='false']/Statement[EmptyStatement]
```

Listing 1.2. XPath query for PMD searching for an `if`-statement that is broken because of an extra semicolon following the condition

```
from x : V{IfStatement}
with not isEmpty(x -->{IfStatementThenStatement}&{EmptyStatement}) and
    isEmpty(x -->{IfStatementElseStatement}&{Statement})
report x end
```

Listing 1.3. GReQL query searching for an `if`-statement that is broken because of an extra semicolon following the condition. See listing 1.2 for the equivalent XPath query for PMD.

AGG even offers a graphical interface for defining graph transformation rules, so no explicit knowledge on graph traversals or query languages is needed. However, as a graph transformation engine, AGG is somewhat oversized for pure matching of graph patterns. Writing code checks as graph transformation rules is hardly intuitive and requires deep understanding of the way, the graph transformation engine is integrated into the grading and tutoring system.

Tools which require to write and compile program code and to reconfigure an existing installation for adding new checks can be considered not appropriate or at least not convenient for learning scenarios with the need for exercise specific checks. The same applies for tools which do not allow any easy extension at all. Query languages like GReQL or XPath are much more appropriate in this scenario, as long as the queries can be passed to the tool individually as needed. Graphical editors may make writing rules easier, but currently no editors specialized on checking rules for static program analysis in learning scenarios exist.

It can be noticed that the differences between syntax trees and syntax graphs as discussed in Section 3.2 are also important for the ease of writing checks. Finding recursive methods can easily be expressed in a graph pattern by two nodes for method declaration and method call, connected by a path from the declaration to the call and an additional access arc from the call to the declaration. Finding the same situation on a syntax tree would require at least string comparison for method names. In addition, finding indirect recursion where `methodA` calls `methodB` and this calls `methodA` again requires additional effort for storing and comparing partial results. In this case, preparing a syntax graph serves as a preprocessing step which performs exactly this additional operations once, so they need not be defined again for every check.

4.3 Weighting Checks

An important issue in automated grading is the design of a marking scheme. Often it is desirable to distinguish between smaller and greater mistakes and to give grades depending on which checking rules have been violated.

CheckStyle, PMD and FindBugs allow weighting by using severity levels for rules. This allows for simple marking schemes where solutions with mistakes of low severity get better grades than solutions with mistakes of higher severity. Constructing more fine grained marking schemes requires additional effort and additional input, providing weights for each checking rule. The data formats used to specify rules in the external tools are not capable of handling these additional information directly.

Graph transformations and GReQL as general approaches for finding patterns in graphs do not offer any native support for weights. As already discussed above, a specific data format for defining rules has to be written anyway, so it is no major additional effort to extend this data format to handle weights.

In summary, dedicated program analysis tools which use severity levels or similar facilities allow to construct simple marking schemes. More general approaches require additional effort even for simple schemes. However, if fine grained schemes with individual weights for every rule are desired, additional effort is necessary in any case.

5 Conclusions

In this paper, several approaches and tools for static source code analysis in automated grading and tutoring tools have been reviewed and compared. It can be stated that it is necessary in learning scenarios to use tools that are able to handle multiple source files. Preprocessing steps, which extend syntax trees to syntax graphs with additional information turned out to be helpful for more flexible and exercise specific configuration of checking tools. Consequently, some of the tools discussed in this paper can be considered insufficient to use the full power of static analysis for feedback generation in e-assessment systems.

Every approach investigated in this paper can be integrated into automated grading and tutoring systems with no major technical obstacles, but additional effort is needed to map fine grained marking schemes to checking rules. Additional effort is unavoidable if general approaches like GReQL or graph transformations should be used, but these approaches do also offer more flexibility towards the integration of customized and exercise specific checks. Consequently, it can be considered acceptable to spent time on this integration work in order to obtain better results and more detailed feedback opportunities.

From these results, a mixture of PMD, GReQL and AGG seems to be the best goal for future development work: It should result in graphical editing of checking rules for multiple source code files based on syntax graphs, focused on static source code analysis and capable of handling fine grained marking schemes. None of the tools discussed in this paper has already reached this level of quality.

References

1. Static Analysis Tool Exposition (SATE 2009) Workshop, Co-located with 11th semiannual Software Assurance Forum, Arlington, VA (2009)
2. AGG website, `http://tfs.cs.tu-berlin.de/agg/`

3. Ala-Mutka, K.M.: A Survey of Automated Assessment Approaches for Programming Assignments. Computer Science Education 15(2), 83–102 (2005)
4. Amelung, M., Forbrig, P., Rösner, D.: Towards generic and flexible web services for e-assessment. In: ITiCSE 2008: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, pp. 219–224. ACM, New York (2008)
5. Bildhauer, D., Ebert, J.: Querying Software Abstraction Graphs. In: Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008), Collocated with ICPC 2008 (2008)
6. Cheang, B., Kurnia, A., Lim, A., Oon, W.-C.: On automated grading of programming assignments in an academic institution. Comput. Educ. 41(2), 121–131 (2003)
7. CheckStyle Project, `http://checkstyle.sourceforge.net`
8. Copeland, T.: PMD applied. Centennial Books (2005)
9. Denny, P., Luxton-Reilly, A., Tempero, E.D., Hendrickx, J.: Understanding the syntax barrier for novices. In: Rößling, G., Naps, T.L., Spannagel, C. (eds.) Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, June 27-29, pp. 208–212. ACM (2011)
10. Douce, C., Livingstone, D., Orwell, J., Grindle, S., Cobb, J.: A technical perspective on ASAP - automated systems for assessment of programming. In: Proceedings of the 9th CAA Conference, Loughborough University (2005)
11. FindBugs Project, `http://findbugs.sourceforge.net/`
12. Gruttmann, S.J.: Formatives E-Assessment in der Hochschullehre. MV-Wissenschaft (2009)
13. Hage, J., Rademaker, P., van Vugt, N.: A comparison of plagiarism detection tools. Technical report, Department of Information and Computing Sciences, Utrecht University (2010)
14. Higgins, C., Hegazy, T., Symeonidis, P., Tsintsifas, A.: The CourseMarker CBA System: Improvements over Ceilidh. Education and Information Technologies 8(3), 287–304 (2003)
15. Hoffmann, A., Quast, A., Wismüller, R.: Online-Übungssystem für die Programmierausbildung zur Einführung in die Informatik. In: Seehusen, S., Lucke, U., Fischer, S. (eds.) DeLFI 2008, 6. e-Learning Fachtagung Informatik. LNI, vol. 132, pp. 173–184. GI (2008)
16. Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O.: Review of recent systems for automatic assessment of programming assignments. In: Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling 2010, pp. 86–93. ACM, New York (2010)
17. Joy, M., Griffiths, N., Boyatt, R.: The BOSS Online Submission and Assessment System. Journal on Educational Resources in Computing (JERIC) 5(3) (2005)
18. Köllmann, C., Goedicke, M.: A Specification Language for Static Analysis of Student Exercises. In: Proceedings of the International Conference on Automated Software Engineering (2008)
19. Leal, J.P., Silva, F.: Mooshak: a Web-based multi-site programming contest system. Software–Practice & Experience 33(6), 567–581 (2003)
20. Mengel, S.A., Yerramilli, V.: A case study of the static analysis of the quality of novice student programs. In: The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education, SIGCSE 1999, pp. 78–82. ACM, New York (1999)

21. Morth, T., Oechsle, R., Schloß, H., Schwinn, M.: Automatische Bewertung studentischer Software. In: Workshop "Rechnerunterstütztes Selbststudium in der Informati", Universität Siegen, 17 (September 2007)
22. Naude, K.A.: Assessing Program Code through Static Structural Similarity. Master's Thesis, Faculty of Science, Nelson Mandela Metropolitan University (2007)
23. PMD Project, `http://pmd.sourceforge.net/`
24. Rutar, N., Almazan, C.B., Foster, J.S.: A Comparison of Bug Finding Tools for Java. In: Proceedings of the 15th International Symposium on Software Reliability Engineering, pp. 245–256. IEEE Computer Society, Washington, DC (2004)
25. Schwieren, J., Vossen, G., Westerkamp, P.: Using Software Testing Techniques for Efficient Handling of Programming Exercises in an e-Learning Platform. The Electronic Journal of e-Learning 4(1), 87–94 (2006)
26. Shah, A.: Web-CAT: A Web-based Center for Automated Testing. Master's thesis, Virginia Polytechnic Institute and State University (2003)
27. Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J.K., Padua-Perez, N.: Experiences with Marmoset: Designing and using an advanced submission and testing system for programming courses. SIGCSE Bull. 38(3), 13–17 (2006)
28. Strickroth, S., Olivier, H., Pinkwart, N.: Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben? In: DeLFI 2011 - Die 9. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. LNI, vol. 188, pp. 115–126. GI (2011)
29. Striewe, M., Balz, M., Goedicke, M.: A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In: Proceedings of the First International Conference on Computer Supported Education (CSEDU), Lisboa, Portugal, March 23-26, vol. 2, pp. 54–61. INSTICC (2009)
30. Striewe, M., Balz, M., Goedicke, M.: Enabling Graph Transformations on Program Code. In: Proceedings of the 4th International Workshop on Graph Based Tools, Enschede, The Netherlands (2010)
31. Truong, N., Bancroft, P., Roe, P.: A Web Based Environment for Learning to Program. In: Proceedings of the 26th Annual Conference of ACSC, pp. 255–264 (2003)
32. Truong, N., Roe, P., Bancroft, P.: Static Analysis of Students' Java Programs. In: Lister, R., Young, A.L. (eds.) Sixth Australasian Computing Education Conference (ACE 2004), Dunedin, New Zealand, pp. 317–325 (2004)
33. Zeller, A.: Making Students Read and Review Code. In: Proceedings of the 5th ACM SIGCSE/SIGCUE Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland, pp. 89–92 (2000)