Static Analysis of Students' Java Programs

Nghi Truong, Paul Roe, Peter Bancroft
Faculty of Information Technology
Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
(n.truong, p.roe, p.bancroft)@qut.edu.au

Abstract

A recent industry survey (Townhidnejad and Hilburn, 2002) has reported that more than fifty percent of a software project's budget is spent on activities related to improving software quality. Industry leaders claim that this is caused by the inadequate attention paid to software quality in the development phase. This paper introduces a static analysis framework which can be used to give beginning students practice in writing better quality Java programs and to assist teaching staff in the marking process. The framework uses both software engineering metrics and relative comparison to judge the quality of students' programs and provide feedback about how solutions might be improved.

Keywords: static analysis, Java, web, tutoring system, XML, online learning.

1 Introduction

Programming is a complex intellectual activity and the core skill for first year IT students. Research has shown that most students are able to write programs; however, their programs are often poorly constructed because they do not consider different solutions to a program. Beginning students often try to solve a problem as quickly as possible without thinking about the quality of their programs (Vizcaino et al, 2000). The study of McGill and Volet (1995) shows that there is a strong relationship between the quality of students' algorithms and the quality of their final programs. The study also reflects that few students adopt a program design methodology when writing a program but rather use one only when required to.

There is a large body of literature which calls for increased emphasis on program design methodologies in introductory programming courses (Townhidnejad and Hilburn, 2002, Sanders and Hartman, 1987, McGill and Volet, 1995, Linn and Clancy, 1992). McGill and Volet (1995) suggest that the best way to improve the quality of students' programs is for instructors to talk through how they solve a specific problem, to discuss alternatives and to allow for backtracking from initial conjectures; however, providing timely feedback on student programming exercises and helping students to think about the quality of their programs are difficult tasks, time consuming and laborious especially with the current large class sizes (Mengel and Yerramilli, 1999). Automated analysis of student programs has the potential to combat this problem. Furthermore, automated analysis may augment the grading process performed by instructors and teaching assistants. More importantly, it can help to give a finer level of detail about the quality of student programs, allowing them more insight towards improving their programming skills.

The contribution of this paper is to describe a static analysis framework for use with beginning students' Java programs. It is designed for both tutoring and semi-automatic assessment purposes. The framework provides feedback about the quality of a student solution, ideas for alternative solutions and their relative merits and hints to improve the student solution. The key features of the framework are its configurability and extensibility. Analyses can be configured to suit different types of exercises. In addition, the complexity of analyses can be controlled by different program abstraction levels. Additional analyses can be plugged into the framework easily. Although the framework can be used as a separate tool, it is particularly useful for "fill in the gap" style exercises such as provided by the Environment for Learning to Program (ELP) (Truong et al, 2002, 2003). At Queensland University of Technology (QUT), the static analysis framework is currently being integrated into the ELP.

The framework brings benefits to both students and teaching staff. It adds intelligent assistance to existing online learning programming environments; thus it increases the level of flexible delivery and facilitates the constructive, effective learning environment of these online learning systems. Although the framework is not able to completely replace the role of instructors or tutors, it helps students to learn in an environment where formative feedback and correct solutions can be obtained immediately and therefore, misconceptions among students are eliminated (Ben-Ari, 2001). Students are able to access as much tuition as they need at their own pace; they are not limited to standard working hours or their current location, by having to come to university to consult teaching staff about their tutorial work. Most importantly, the feedback provided by the framework helps students to justify their choice of algorithms for solving a problem; making them to become more effective programmers (Sanders and Hartman, 1987).

With the analysed result from the framework, the marking task will be less time consuming and laborious.

This paper is organized into six sections. Previous systems that have been developed to help students learn to program are discussed in Section 2. An overview of the ELP system is described in Section 3. Section 4 of the paper gives a general overview of the static analysis framework. The current implementation is reported in Section 5. Lastly, limitations and the future development plans for the framework are discussed in Section 6.

2 Approaches and Systems

As previously mentioned, the framework described in this paper can be used for both tutoring and semi-automatic marking purposes. This section gives an overview of research into automatic programming tutors and marking systems. It also describes systems which have had a major impact on the design and implementation of the framework: Talus (Murray, 1988), CourseMaster (CourseMaster, 2000) and Expresso (Hristova et al, 2003).

2.1 Approaches

Static analysis is the process of examining source code without executing the program. It is used to locate problems in code including potential bugs, unnecessary complexity and high maintenance areas. Dynamic analysis is the process of running a program through a set of data. The main aim of dynamic analysis is to uncover execution errors and to help evaluate the correctness of a program. Applications in tutoring and automatic marking make use of either static analysis or dynamic analysis or both to evaluate student programs. There are many techniques to implement static analysis; however, approaches that have been adopted in computer science education applications vary from string matching based on the program source (simplest form) to matching program graph representations (complicated form).

Much research has been devoted to developing a system to help novice students learn to program. According to Deek and McHugh (1998), a large part of this research has focused on issues concerning syntax and has not addressed the lack of problem solving skills and analysis and design methodologies among beginning students.

Software metrics is one well known way to measure the quality of programs. Despite that, few of the existing systems have adopted metrics to evaluate student programs (Mengel and Yerramilli, 1999). Leach and Mengel (1995, 1999) claims that Halstead metrics (Halstead, 1977), McCabe cyclomatic complexity (McCabe, 1976), number of coupling instances and Berry and Meekings style guide line (Harrison and Cook, 1986) are common and useful static metrics for computer science education applications. However, they are often used for marking and plagiarism detection purposes rather than for teaching students design and writing good quality programs which require more detail feedback compare to the other two purposes.

Automatic grading systems are economical and effective. This kind of system reduces the workload for instructors and improves the student's learning experience by providing instant feedback. Because of these benefits, widespread research has been carried out to develop automatic grading systems, the idea being introduced by Hollingsworth (1960). Among the earliest systems were GRADER1 and GRADER2 used at Stanford University with beginning students' BALGOL programs (Forsythe, 1964). A student program can be assessed in various ways which include style, correctness, efficiency and plagiarism. Examples of systems that perform only static analysis are ASSYST (Jackson and Usher, 1997), CAP (Schorsch, 1995) and Expresso (Hristova et al, 2003). A system that performs only dynamic analysis is TRY (Reek, 1989). There are systems which integrate both tutoring and automatic marking to develop courseware, for example CourseMaster (CourseMaster, 2000) and BOSS (Joy and Luck, 1998).

The goal of the program analysis framework described in this paper is to use software engineering metrics tools and good programming practices to judge the quality of student programs. The framework performs the analysis based on XML representation of program abstract syntax trees; it incorporates both quantitative and qualitative analyses to provide detailed feedback to students.

2.2 Systems

Talus (Murray, 1988) is an automatic program debugging tool for the Lisp language. Talus diagnoses both non-stylistic and stylistic bugs at three different levels of abstraction including algorithm level, function level and implementation level. It uses a plan based program analysis approach and debugs input programs in four steps: program simplification, algorithm recognition, bug detection and bug recognition. Program simplification transforms the input program into a Lisp code dialect. In algorithm recognition, the simplified functions are parsed into frames and partially matched frames in the task representation. Once it identifies bugs in the input program, Talus attempts to correct them using techniques based on theorem proving and heuristic methods. Talus has three main limitations. Firstly, it can only analyze programs with functions that are allocated exactly as specified in the programming plan. This is a serious limitation when dealing with large programs. Secondly, it assumes the task is already known. Lastly, Talus provides only limited data structure definitions and has problems with large programs and imperative programming style (Song et al, 1996).

CourseMaster (CourseMaster, 2000) is a client server system for delivering course based programming. It provides functions for automatic assessment of students work in Java and C++ and administration of the resulting marks, solutions and course materials. A student is able to develop a program, submit it to the server for marking or evaluation and get instant feedback. The student program is analyzed for typographic layout, dynamic execution, program features, flowchart, object oriented design and logic circuit marking. The analysis process relies heavily on the standard Unix C utility, `lint`. Thus, the main

drawback of the system is that it is not platform independent.

Expresso (Hristova et al, 2003) is designed to identify beginning student Java programming errors. Expresso detects students' Java syntax, semantic and logic errors and provides hints about how the problem should be fixed. The input program removes comments and white spaces and tokenizes the text into small tokens. Expresso then uses string matching techniques to detect mistakes and generate feedback. Feedback messages generated by the tool are enhanced compiler error messages.

3 ELP

ELP is an online interactive and constructive environment for learning to program, which is currently being developed at QUT to help Information Technology students to write Java programs successfully at an early stage in their learning. Students undertake web based programming exercises from the ELP web server. They complete exercises and submit them to the server for compilation. If there are no syntax errors in the student's solution, the resulting class files of the exercise are packed together with other necessary libraries in a JAR file and subsequently downloaded and run on the student's machine. Otherwise, a compilation error message is returned. All exercises in the ELP system are "fill in the gap" exercises. This type of exercise not only reduces the complexity of writing programs but also allows students to focus on the problem to be solved. Figure 1 illustrates the integration between the ELP system and the program analysis framework that is described in this paper.

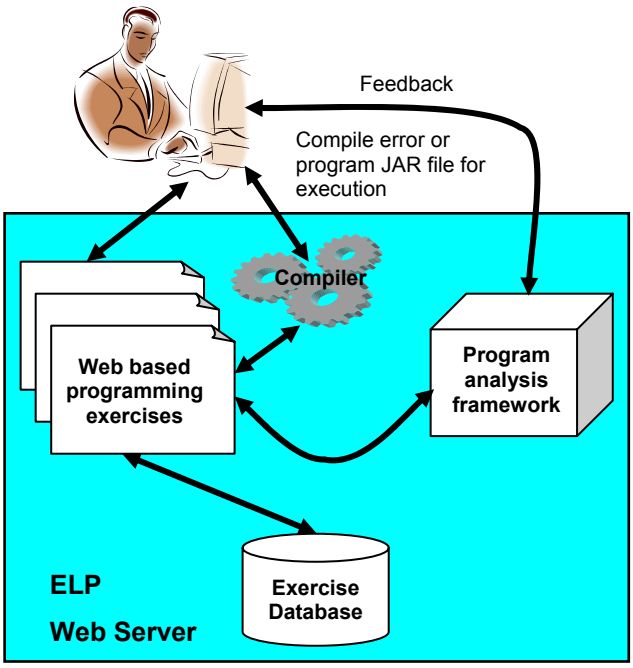

Figure 1: ELP and the program analysis framework integration

4 Framework Design

This section describes the design of the static analysis framework. Common mistakes among beginning Java programming students at QUT are discussed in Section 4.1. These mistakes play an important role in the design of the framework. Section 4.2 gives an overview of the analyses that the framework currently provides.

4.1 Students' Common Java Errors

In order to identify students' programming practices and their well-known logic errors, a comprehensive literature review was carried out. Subsequently, a survey of students' work was conducted in the Faculty of Information Technology at QUT to validate the literature review findings and to gain a better understanding of mistakes that beginning students often make.

The literature indicates that most of the previous research was conducted on a very small scale. The work of Hristova (2003) is one of the few large scale surveys. The survey was conducted among Java teaching staff at Bryn Mawr College and 58 teaching staff from 58 schools in the United States. Sixty-two Java programming errors were reported; however 20 of these are considered most important and they were grouped as follows: syntax errors, semantic errors and logic errors. Various other resources on the web identifying common student Java programming errors include (Topor, 2002, Ziring, 2001).

A survey was conducted among teaching staff and students of an introductory programming course in the Faculty of Information Technology at QUT. The course aims to teach students basic programming using Java as well as some object oriented concepts. "Java: A Framework for Programming and Problem Solving" (Lambert and Osborne, 2002) is used as the textbook. Students are required to design, implement, execute and debug small Java programs. The results of the survey revealed nine common poor programming practices and five common logic errors that occurred in beginning students' programs.

Table 1 summarizes the findings of the literature and the survey.

Poor Programming Practices
• Too many loop and conditional statements
• Not enough methods
• Use of global variables rather than parameters to a method
• Too large methods
• Use of magic numbers (literals)
• Unused variables
• Perform unnecessary checking with Boolean expression
• Un-initialised variables
• Inappropriate access modifiers

Common Logic Errors
• Omitted "break" statement in a case block • Omitted "default" case in a switch statement • Confusion between instance and local variables • Omitted call to super class constructor

Table 1: Beginning students common errors

4.2 The Static Analysis

The static analysis process was designed with the main aim of judging the quality of students' programs. It can be used to help beginning students learn to program and to provide teaching staff with semi-automatic marking tools. As mentioned earlier, all exercises in the ELP system are "fill in the gap" exercises therefore only the gap code supplied by the student is analysed. Although a gap can be any number of missing lines in an exercise on the ELP system, only well formed gaps are analysed by the framework to ensure that there is enough information about the context. Examples of well formed gaps are a statement or block of statements, a method or a complete class. It is important to point out that the framework analyses only compilable programs.

Since the framework only analyses small programs, our main conjecture is that a program's structure reflects its quality. As a result of that, analyses that are provided by the framework only focus on the structure and quality of code. It is important to make the distinction between structural analysis and semantic analysis. While structural analysis emphasizes the design of programs, semantic analysis is often used in program optimization and verification.

The two main design aims of the framework are configurability and extensibility. Analyses are provided as a set of functions and instructors can specify which analyses should be carried out for each gap in an exercise. These analyses make use of dynamic loading at run time so that other additional analyses can be easily plugged in, if required. There are two distinct groups: software engineering metrics analysis and structural similarity analysis, described in Sections 4.2.1 and 4.2.2 respectively.

4.2.1 The Software Engineering Metrics Analysis

Software metrics is a well-known quantitative approach used to measure software quality. This analysis is based on software complexity metrics and good programming practice guide lines to assess the quality of student solutions. Cyclomatic complexity, which measures the number of linearly-independent paths through a program module, is adopted in the framework because it provides useful information about the structure of a program.

Other software engineering metrics have been used to evaluate beginning student programs, for example Halstead software metrics were used in (Leach, 1995) to detect plagiarism. Coupling and cohesion metrics and Berry-Meekings style guideline metrics were used in Jackson (1996). These software engineering metrics can be easily loaded into the framework at runtime if desired because of its extensibility characteristic.

4.2.2 Structural Similarity Analysis

The purpose of this analysis is to refine the result of the software engineering metrics analysis and to check how the structure of the student solution compares with model solutions. In the analysis, the student solution and model solution are both transformed to an abstract pseudo code form which represents just the abstract algorithmic structure of the programs. The abstract representations of the student solution and model solution are compared to identify differences. Feedback to both students and instructors indicates the similarity of the student and model solutions. It is important to note that the techniques that are used to design this analysis only work for simple introductory programs.

By comparing student solutions with model solutions, the framework is able to identify high complexity areas in the student code, such as lengthy methods. Unmatched areas between student solutions and model solutions can be used to predict and provide better feedback to students if their solutions result in an incorrect output in dynamic analysis. Thus structural similarity analysis closes the gap between static analysis and dynamic analysis which exists in earlier related research.

Rich and Wills (1990) raised several issues with the use of cliché matching including syntactic and implementation variation; thus it is difficult to anticipate all possible solutions for a problem. To overcome this drawback, the framework is designed so that when the system cannot find a match between the student solution and all available model solutions, the student solution is sent to teaching staff for review. If the instructor recognizes that it is another allowable solution for the exercise, it can be added to the model solution list. In addition, as only small or "fill-in the gap" exercises are analysed by the framework, the implementation variation is very small. Last but not least, the matching process in the framework is based on the algorithm structure instead of exact match.

In order to ensure the framework may be used effectively with different types of exercises, the abstraction and matching processes are configurable by instructors to suit the individual exercise. For example the abstract pseudo code form can retain detailed information such as variable names and method calls or just statistics of the code. Similarly, the matching process varies from exact to relative matching of the statistical information.

5 Framework Implementation

The software engineering metrics and structural correctness analyses operate on the program Abstract Syntax Trees (AST). The AST is represented using XML. When a gap exercise is submitted for analysis, it is first converted to an XML marked-up AST using the ANTLR

(Parr, 2003) parser. The student solution is analysed for all options in the software engineering metric as specified by the instructor and feedback is generated. After that the abstraction of the model solution AST and student AST are obtained from the program transformation process. These two abstraction documents are compared with each other to identify differences and provide further feedback to students. Figure 2 illustrates the overview of the static analysis process.

Section 5.1 discusses the usage of AST and XML to implement the static analysis. The detailed implementation of each analysis is described in Section 5.2 and 5.3.

5.1 Implementation Consideration

An AST representation was chosen as the base type to perform the analysis because according to Badros (2000), it can efficiently exploit a well-defined and well understood structural representation of a program. This will enable the framework to give more detailed feedback about the quality of student programs.

The framework makes use of XML extensively. As well as the analysis performed on the XML marked-up representation of a program, the results of the analysis and the configuration are also XML documents. The use of XML has brought several advantages to the framework including: easy to understand and manipulate, extensible, widely supported and human readable (Mamas and Kontogiannis, 2000).

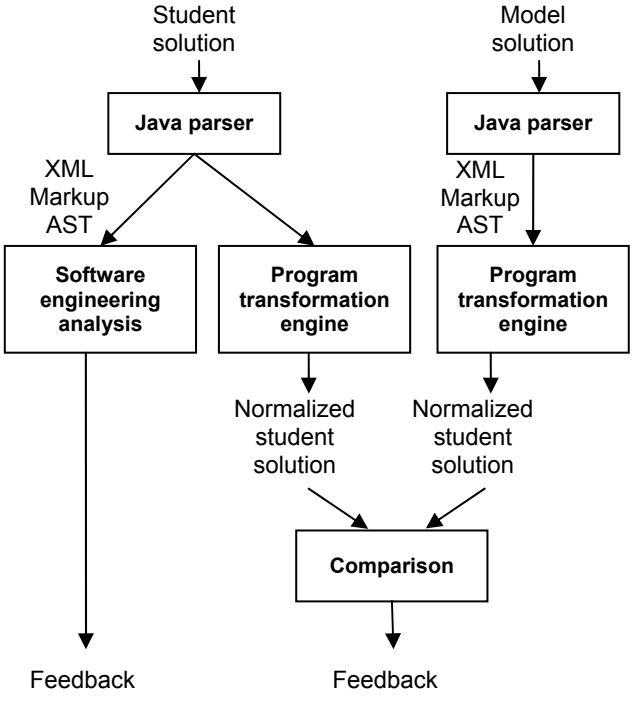

Figure 2: An overview of Static Analysis

5.2 Implementation of Software Engineering Metrics Analysis

Currently, the system provides a set of configuration functions to check students' common poor programming practices and logic errors mentioned in Section 4.1. The key point in this analysis is that the functions are configurable for each gap in an exercise. Table 2 lists all available functions together with their descriptions.

All analyses are stored in the "StaticAnalysis" folder on the server and are only loaded when they are specified as one of the required analyses for a gap. A new analysis which can be a Java class file or JAR package can be added to the framework easily by saving it to the located folder. The only requirement for the new component is that it needs to implement the `StaticAnalysis` interface which is show in Figure 3.

```
public interface StaticAnalysis {
   public String getShortDes();
   public String getLongDes();
   public Document analyse(
        Element gap, Document configDoc,
        Document solution);
   public Document similarity(
        Document studSol, Document modelSol,
        Document configDoc);
}
```

Figure 3: StaticAnalysis Interface

Check	Description
Program Statistics	Count the total number of variables, statements and expressions in a gap.
Shadow Variables	Check if a variable is declared in both class scope and method scope.
Cyclomatic Complexity	Count the number of logic decisions in a program.
Unused Parameters	Check if there are any unused parameters in a method.
Redundant Logic Expression	Detect redundant logical e.g. expressions "x==true".
Unused Variables	Check if there are any unreferenced variables in a specified scope.
Magic Numbers	Ensure student solutions do not have hard coded numbers or string literals.
Access Modifiers	Ensure variables and methods have the correct modifiers.
Switch Statements	Ensure that all switch statements have "default" case and in each case block there is a "break" statement.
Character Per Line	Calculate number of characters per line (max 80).
No Tabs	Ensure that space is used to indent the code rather than Tab key.

Table 2: Functions provided

Java reflection is used to load and invoke analyses at run time. The `analyse` method of the class which implements the `StaticAnalysis` interface will be invoked for all software engineering metrics analyses whereas the `similarity` method will be invoked for all structural similarity analyses. The `getShortDes` and `getLongDes` methods are used to display descriptions for the analysis.

Feedback to students can be either automatically generated or customized by instructors. The feedback received specifies the line number in the solution where the poor code lies, together with suggestions of how the solution might be improved. In the future, it is planned to generate feedback to aid tutors with marking.

5.3 Implementation of Structural Similarity Analysis

As with the software engineering metrics analysis, the structural similarity analysis is only loaded when it is specified as required for an exercise. In this analysis, one or more model solutions and the student solution for a gap are transformed into a simpler form and compared with each other. If a student solution has a matching structure, a congratulatory message is returned. Otherwise feedback highlighting all the differences between the student and model solutions together with instructors' suggestions of how the problem should be solved is given. These suggestions are embedded in the XML mark up of the exercises.

Program abstraction is achieved by adding generic nodes to the AST. For example a generic loop node is used to represent any form of loop. Similarly, there are generic expression and selection nodes. Other generic nodes represent statement counts. Figure 4 illustrates a gap for a block of statements and its normalized form. This normalization process also helps to limit the variation of possible solutions for a problem.

```
<gap>
<statements>
   <assignment>1</assignment>
   <methodCall>1</methodCall>
   <loop>
      <condition>
         <trueBranch>
            <methodCall>1</methodCall>
         </trueBranch>
         <falseBranch>
            <methodCall>1</methodCall>
         </falseBranch>
      </condition>
      <assignment>1</assignment>
      <methodCall>1</methodCall>
   </loop>
</statements>
</gap>
```

```
guess = reader.readInt("Guess a number " +
               "between 1 and 100 ");
while(guess != secret){
   if(guess < secret){
      writer.println("Your guess is low");
   }
   else {
      writer.println("Your guess is high");
   }
   guess = reader.readInt("Guess a " +
        "number between 1 and 100 " );
}
```

Figure 4: A gap and its normalization

5.4 Example

The following example illustrates how the framework integrates into the ELP system.

Question:

Write a simple program that obtains two integer values – lowerLimit and upperLimit from the user. Display all integers between lowerLimit and upperLimit in ascending order.

Figure 5 illustrates a "fill in the gap" ELP version of this exercise with a student solution in the gap. The underlined statements in the gap show the differences in the student solution and the model solution.

A student submits an exercise to the server for analysis by pressing the "Analyse" button. The framework builds the complete Java source file and compiles the student solution to ensure that there are no syntax errors. If the compilation process is successful, the whole Java source file is run through the customized ANTLR parser to obtain the XML marked-up AST representation of the program. A GapExtractor engine processes the resulting AST to extract the gap. It then extracts the AST that represents the student solution from the complete program AST. Figure 6 below represents the static analysis configuration together with the model solution for the gap.

With reference to Figure 6, all XML elements that are children of SoftwareEng node (CyclomaticComplexity, CheckRedundantLogicExpression) are named to match the corresponding Java class. As mentioned earlier, these classes implement the StaticAnalysis interface. The framework reads the analysis configuration for the gap and uses Java reflection to invoke the analyses.

With the structural similarity analysis, the skeleton which is extracted from the marked-up exercise, the exercise solutions which are constructed from possible solutions embedded in each gap and the AST marked up XML for the exercise solution are generated and stored on the server the first time the exercise is analysed. Unlike the software engineering metrics analysis, this analysis is class based. If an exercise has more than one gap, all gaps need to be completed in order to carry out the analysis. When an exercise has more than one class, depending on

the dependency among classes, students might need to complete all classes in the exercise. If the gap has more than one solution, they are arranged sequentially in the marked-up exercise. The `similarity` method will be invoked for all analyses that belong to the structural similarity analysis.

```
import TerminalIO.*;

public class SafeCountBy1
{
    KeyboardReader reader =
                        new KeyboardReader();
    ScreenWriter writer =
                        new ScreenWriter();

    public void run()
    {
        writer.println("Welcome to the " +
                       "SafeCountBy1 program");

        //Input variables
        int lowerLimit;
        int upperLimit;

        //Intermediate variables
        int counter;

        //Read lower and upper limit
        lowerLimit =
             reader.readInt("lower limit: ");
        upperLimit =
             reader.readInt("upper limit: ");

       counter = lowerLimit;
       while(((counter <= upperLimit)== true)
                && (counter >=0))
       {
          writer.println("counter = " +
                                        counter);
          counter = counter + 1;
       }
     }
     public static void main(String[] args)
     {
        SafeCountBy1 tpo = new SafeCountBy1();
        tpo.run();
     }
}
```

Save | Compile & Save | Reset | Analyse

Figure 5: An ELP exercise example with a student solution

The ELP system displays the results of the analysis as a list of links presented to the student; Figure 7 illustrates the static analysis feedback returned to student. The student can select which of the analysis they would like to see. Each analysis has a long and a short description; the short description is displayed as a tool tip for the link; the student can view the long description by clicking on the "View Description" button. As shown in the `StaticAnalysis` interface, all the analyses return an XML document which represents the results of the analysis. When the student selects an analysis to view, a servlet that belongs to the analysis processes the result document to generate feedback.

```
<Gap>
<Analysis>
   <Static>
      <SoftwareEng>
           <CyclomaticComplexity/>
           <CheckRedundantLogicExpression/>
      </SoftwareEng>
      <StructuralSimilarity/>
   </Static>
</Analysis>
<Solution>
while(lowerLimit < upperLimit){
   writer.println("Sorry, lower limit may " +
       " not be greater than upper limit!");
   upperLimit = reader.readInt(" upper " +
                                  " limit ");
}
counter = lowerLimit;
while(counter <= upperLimit){
   writer.println("counter = " + counter);
   counter = counter + 1;
}
</Solution>
</Gap>
```

Figure 6: The gap analysis configuration and solution

Save | Compile & Save | Reset | Analyse

Static Analysis Result

Cyclomatic Complexity	View Description
Redundant Logic Expression	View Description
Structural Similarity	View Description

Figure 7: Static Analysis Result

The value computed by the CyclomaticComplexity analysis is obtained by counting the number of logic decisions in the code plus one. For example, the cyclomatic complexity value will be three for the student solution. The feedback can be either only the complexity value such as in the given example or customized by comparing the specified accepted value and its variation depending on the configuration. With the CheckRedundantLogicExpression analysis, the feedback is a list of logic expressions that perform redundant checks in the code. In this example, `(counter <= upperLimit) == true` is returned.

The structural similarity analysis feedback consists of the comparison between the suggested model solution and

structure of the student solution. Figures 8 and 9 illustrate the Structural Similarity analysis feedback and the suggested solution. Through the feedback shown in Figure 8, the student can recognize that they have missed one loop in their gap solution.

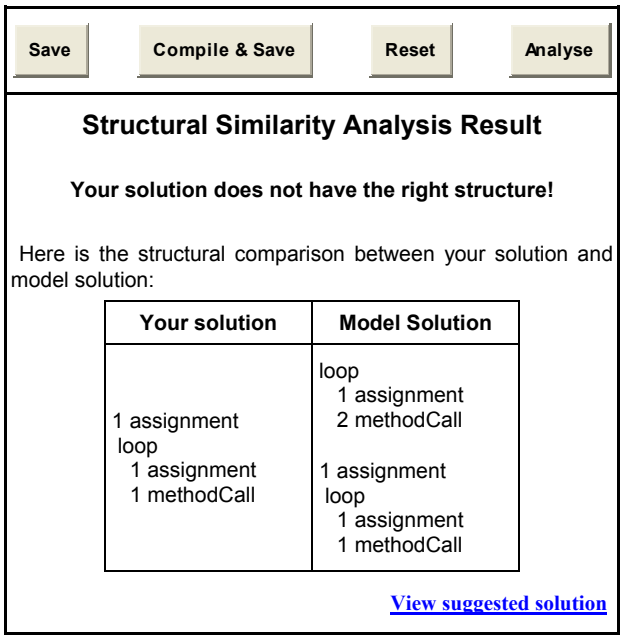

Figure 8: Structural Similarity analysis Feedback

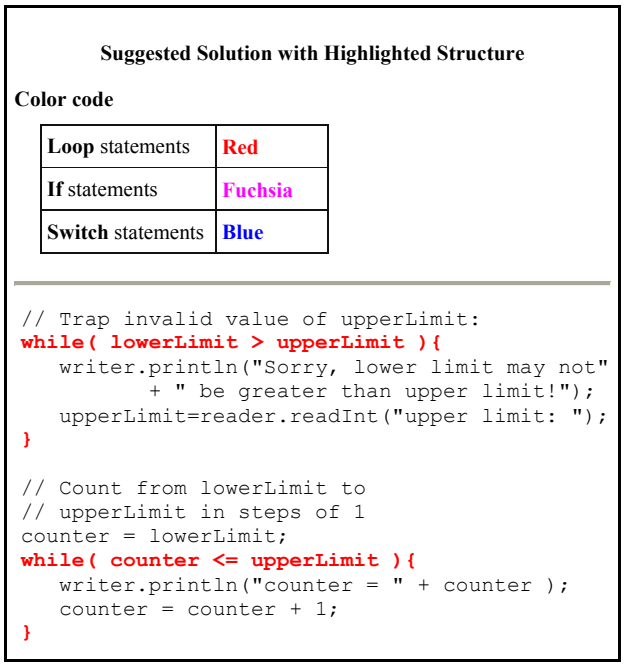

Figure 9: Suggested solution with code highlighted

6 Conclusions and Future Work

The static analysis framework consists of two analyses: software engineering metrics and structural similarity. The first evaluates the quality and the second examines the similarity in structure of student programs compared with model solution. The analyses are performed on XML marked-up AST representations of programs. Feedback to students includes comments about the quality and structure of their programs, hints of how the solution might be improved and alternative solutions.

Overall, the framework has four limitations. First, the chosen technique only works with small or "fill in the gap" type programming exercises to minimize the implementation variation in structural similarity analysis. Second, the framework is able to analyse only well-formed gaps. Third, the framework does not implement semantic analysis; however, with its extensible architecture, additional analyses can be plugged in easily. Last, the framework only analyses syntactically correct programs. All gaps need to be completed in order to carry out the analysis with multiple dependent gaps exercises.

An evaluation of the framework in a class of 400 students has been re-scheduled for first semester 2004 to coincide with the introductory programming course at QUT. However, the framework was designed and tested on student tutorial exercises over the last few semesters. In addition, it is being continuously evaluated by teaching staff in the faculty and consistently receives positive feedback.

7 References

Badros, G. J. (2000): JavaML. http://www.cs.washington.edu/homes/gjb/JavaML/. Accessed March 2002.

Ben-Ari, M. (2001): Constructivism in Computer Science Education. *Journal of Computers in Mathematics & Science Teaching* **20**(1): 24-73.

CourseMaster: School of Computer Science & IT, The University of Nottingham, UK. http://www.cs.nott.ac.uk/CourseMaster/cm_com/index.html. Accessed 2002.

Deek, F. and McHugh, J. (1998): A survey and critical analysis of Tools for Learning Programming. *Journal of Computer Science Education,* **8**(2): 130-178.

Forsythe, G. E. (1964): Automatic machine grading programs. *Proc. the 1964 19th national conference*, 141-401, ACM Press.

Halstead, M. H. (1977) *Elements of software science,* Elsevier, New York.

Harrison, W. and Cook, C. R. (1986): A Note on the Berry-Meekings Style Metric. *Communications of the ACM,* **29**(2): 132-125.

Hollingsworth, J. (1960): Automatic graders for programming classes. *Communications of the ACM,* **3**(10): 528-529.

Hristova, M., Misra, A., Rutter, M. and Mercuri, R. (2003): Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. *Proc. the 34th SIGCSE technical symposium on Computer science education,* Reno, Nevada, USA, **34**:153-156, ACM Press.

Jackson, D. (1996): A software system for grading student computer programs. *Computers Education,* **27**(3/4): 171-180.

Jackson, D. (1997) A software system for grading student computer programs. *Proc.* the twenty-eighth SIGCSE technical symposium on Computer science education, San Jose, California, United States **28**: 335-339, ACM Press.

Joy, M. and Luck, M. (1998): Effective electronic marking for on-line assessment. *Proc. The 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education.* Dublin City University, Ireland, 134-138, ACM Press.

Lambert, K. and Osborne, M. (2002) *Java: A Framework for Programming and Problem Solving,* Brooks/Cole.

Leach, R. J. (1995): Using metrics to evaluate student programs. *ACM SIGCSE Bulletin,* **27**(2): 41-43.

Linn, M. C. and Clancy, M. J. (1992): Can experts' explanations help students develop program design skills? *International Journal Man-Machine Studies,* **36**(4): 511-551.

Mamas, E. and Kontogiannis, K. (2000): Towards Portable Source Code Representations Using XML. *Proc. Seventh Working Conference on Reverse Engineering,* Brisbane, Australia, **7:**172-182, IEEE.

McCabe, T. J. (1976): A Complexity Measure. *IEEE Transactions on Software Engineering,* **2**(4): 308-320.

McGill, T. and Volet, S. (1995): An Investigation of the Relationship between Student Algorithm Quality and Program Quality. *SIGCSE Bulletin,* **27**(2): 44-48.

Mengel, S. and Yerramilli, V. (1999): A Case Study Of The Static Analysis Of the Quality Of Novice Student Programs. *Proc. Thirtieth SIGCSE technical symposium on Computer science education*, New Orleans, Louisiana, United States, **13:**78-82.

Murray, W. M. (1988) *Automatic Program Debugging for Intelligent Tutoring Systems,* Morgan Kaufmann, Pitman, London.

Parr, T.: ANTLR, http://www.antlr.org. Accessed 2002.

Reek, K. A. (1989): The TRY system -or- how to avoid testing student programs. *Proc. The twentieth SIGCSE technical symposium on Computer science education*, Louisville, Kentucky, United States, **21**:112-116, ACM Press New York, NY, USA.

Rich, C. and Wills, L. M. (1990): Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software,* **7**(1): 82-89.

Sanders, D. and Hartman, J. (1987): Assessing the quality of programs: A topic for the CS2 course. *Proc. Eighteenth SIGCSE technical symposium on Computer science education*, St. Louis, Missouri, United States, **19:**92-96, ACM Press.

Schorsch, T. (1995): CAP: An automated self-assessment tool to check Pascal programs for syntax, logic and style errors. *Proc. The twenty-sixth SIGCSE technical symposium on Computer science education*, Nashville, Tennessee, United States, 168-172, ACM Press.

Song, J. S., Hahn, S. H., Tak, K. Y. and Kim, J. H. (1996): An Intelligent tutoring system for introductory C language course. *Computers Education,* **28**(2): 93-102.

Topor, R. W.: CIT1104 Programming II: Common (Java) programming errors, http://www.cit.gu.edu.au/~rwt/p2.02.1/errors.html. Accessed 1 May 2002.

Townhidnejad, M. and Hilburn, T. B. (2002): Software Quality Across the Curriculum. *Proc. The 15th Conference on Software Engineering Education and Training*, Covington, KY, USA, **15:**268-272, IEEE.

Truong, N., Bancroft, P. and Roe, P. (2002): ELP - A Web Environment for Learning to Program. *Proc. The 19th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education*, Auckland, New Zealand, **19:**661-670.

Truong, N., Bancroft, P. and Roe, P. (2003): A Web Based Environment for Learning to Program. *Proc. Twenty-Sixth Australasian Computer Science Conference*, Adelaide, **16:**255-264.

Ziring, N.: Java Mistakes Page, http://users.erols.com/ziring/java-npm.html#item9. Accessed Sept 2002.