Adding Functionality to PMD Java Checker

Robbie Hott

May 10, 2006

1 Introduction

Software Engineering has shown us that it is much cheaper to find errors in code as early as possible, and that found early, they are much easier to fix [3]. This has led to the outcrop of many compile-time checks for finding bugs early: tools, environments, and methods. Software Engineers use formal methods such as Cleanroom development, in which they develop code without compiling it to stamp out bugs, Rapid Prototyping, in which they build a quick model knowing they will throw it away, and Theorem Proving, proving certain properties of the system before coding. A leader in this area is Praxis [1]. The tools Software Engineers use include ESC/J, Java PathFinder, PVS for proof verification, Alloy for model checking, and others. However, most of these tools require a learning curve, or at least an experienced programmer to use, but a similar tool, PMD, would allow for use earlier as a programmer [5].

2 Background

PMD allows the programmer to write rules that the code must follow in order for the code to be considered correct by PMD. These rules can be written in one of two formats, XML or Java classes. It can allow this because it uses a JavaCC parser to parse the file into an Abstract Syntax Tree, as can be seen in Figure 2. Therefore, that tree can be browsed by either XML or Java. PMD already includes many rules, including some to ensure curly braces follow a `while` statement, that there are no unnecessary `if` statements, and some that ensure that the code is formatted correctly, just to name a few. It also can be integrated into editing environments, such as Eclipse, and has a GUI for easy use with any level of coder.

This provides a powerful framework in which the checking of code can be enhanced. PMD and these other tools have come a long way, but further work on them can help coders find bugs earlier and avoid the costly time of bug-hunting later on.

3 Related Work

Other programs are available that are doing similar work. This includes ESC/Java [4]. ESC/Java and many other programs that check code at compile time require that properties and rules about the code are inserted into the code in a language form known as JML, Java Markup Language. Right now, ESC/Java is in version 2, but there is not a large amount of information available about it.

```
class MyClass {
   Vector v;
   Integer a;

   int main() {
      for (int i = 1; i < 25; i++) {
         int b = 5;
         int c = 56;
         b = b + 34 + c;
      }
   }
}
```

Figure 1: Small example of Java Code to create an AST from using PMD.

Amme, Dalton, Ronne, and Franz's SafeTSA is a Java VM and compiler replacement that uses a different form of byte code, which is smaller than Java Byte Code, however it is mainly used in academia [2]. Their method for ensuring type-correctness is to use Static Single Assignment, which helps to reduce the code verification effort at the consumer side. As of late, their website `safetsa.org` has disappeared. This would have made it more difficult to find SafeTSA, which would need to be updated to accept Java 1.5.

4 Improvements and Results

Many improvements were made in PMD by writing rulesets that add functionality to PMD. These took two basic directions: type checking and simple rule handling.

4.1 Type Checking

The first ruleset attempted was a ruleset to check type issues. The typechecking ruleset was written to include explicit casts and implicit casts through function calls. The function call rule code is run every time the AST encounters a function declaration, and the parameters are researched. The explicit cast rule code is run every time the AST encounters a Cast expression. Both rules look back through the tree and attain the original declaration of the variable and function call rule checks the original definition of the function. They are compared and PMD outputs a warning if one is needed.

This ruleset worked for all of the simple cases, but was too complex to run on the larger test cases. However, this set is mainly used to give the programmer a heads up because the Java javac compiler will warn the user of loss of accuracy or unacceptable calling of functions with improper types. Downcasting is a normal procedure for Java, especially in older version when adding to a Collection class provided by Java. Therefore, the casting functionality is useful in large software that an updater might want to see how variables are being downcast, but not useful for the small programmer that must downcast to use Java's built in list and collection functionality.

```
CompilationUnit
 TypeDeclaration
  ClassOrInterfaceDeclaration(MyClass)(class)
   ClassOrInterfaceBody
    ClassOrInterfaceBodyDeclaration
     FieldDeclaration:(package private)
      Type
       ReferenceType
        ClassOrInterfaceType:Vector
      VariableDeclarator
       VariableDeclaratorId:v
    ClassOrInterfaceBodyDeclaration
     FieldDeclaration:(package private)
      Type
       ReferenceType
        ClassOrInterfaceType:Integer
      VariableDeclarator
       VariableDeclaratorId:a
    ClassOrInterfaceBodyDeclaration
     MethodDeclaration:(package private)
      ResultType
       Type
        PrimitiveType:int
      MethodDeclarator:main
       FormalParameters
      Block
       BlockStatement
        Statement
         ForStatement
          ForInit
           LocalVariableDeclaration
            Type
             PrimitiveType:int
            VariableDeclarator
             VariableDeclaratorId:i
             VariableInitializer
              Expression
               PrimaryExpression
                PrimaryPrefix
                 Literal:1
          Expression
           RelationalExpression:<
            PrimaryExpression
             PrimaryPrefix
              Name:i
            PrimaryExpression
             PrimaryPrefix
              Literal:25
          ForUpdate
           StatementExpressionList
            StatementExpression
             PostfixExpression:++
              PrimaryExpression
               PrimaryPrefix
                Name:i
          Statement
           Block
            BlockStatement
             LocalVariableDeclaration
              Type
               PrimitiveType:int
              VariableDeclarator
               VariableDeclaratorId:b
               VariableInitializer
                Expression
                 PrimaryExpression
                  PrimaryPrefix
                   Literal:5
            BlockStatement
             LocalVariableDeclaration
              Type
               PrimitiveType:int
              VariableDeclarator
               VariableDeclaratorId:c
               VariableInitializer
                Expression
                 PrimaryExpression
                  PrimaryPrefix
                   Literal:56
            BlockStatement
             Statement
              StatementExpression
               PrimaryExpression
                PrimaryPrefix
                 Name:b
               AssignmentOperator:=(simple)
               Expression
                AdditiveExpression:+
                 PrimaryExpression
                  PrimaryPrefix
                   Name:b
                 PrimaryExpression
                  PrimaryPrefix
                   Literal:34
                 PrimaryExpression
                  PrimaryPrefix
                   Name:c
```

Figure 2: AST generated from the Java code in Figure 1.

4.2 Simple Rule Handling

A ruleset and supplementary material to handle that ruleset was written in Java to handle an external file as input to PMD. The file allows a user to declare methods as pure or variables that should not change in certain methods. A sample file can be seen in Figure 3. The ruleset reads in this sample file and checks at each class to see if there is a method that is declared pure and if so, checks that no variables created outside the method are changed within the method. Similarly, if a variable is declared unchanged in a method, the method is checked to see if that variable is assigned something different. If the variable has a function called on it, we cannot guarantee that the variable is unchanged, so we will note that a function has been called on it.

Simple rule files were written and this ruleset was run on sample student code from an undergraduate Computer Science course. This method correctly identified impure methods and methods that changed the variables given it.

```
pure MyClass:main
pure MyClass:getData
pure MyClass:getInfo
unchanged MyClass:getInfo:a
unchanged MyClass:getInfo:c
```

Figure 3: Sample simple rule file used to test the code in Figure 4.

5 Future Direction

The next big step for PMD is to include other files along with the files it is testing for a complete system test. That can be done by including the imported files of each source file into the files to check. I have started to implement this into PMD, however, as systems grow, the java files might not exist on a given system, such as when the Java packages are imported. That leads to the need to parse class files, which becomes even more complicated. Once this has been implemented in PMD, it will fix many cross-class bugs and issues in PMD and allow the tool to be much more comprehensive and powerful.

6 Conclusion

It turns out that the Simple Rule handling to check for pure and unchangeable variables is very useful. It allows code to be checked by coders or testers without the declarations for the testing required to be in the Java source file. Coders can write the code and the separate file with pure and unchanged declarations and keep the Java source code clean of these checks. Similarly, testers can use the file to ensure the code is correct without the coder seeing the tests run or having the coder write the tests ahead of time. This is very helpful in the classroom setting, allowing a professor to write tests for a structure given to students, and can check whether functions such as `getData()` accidently change `Vector data` in the process when a student comes in for help. It also allows the students to easily perform these checks as well. It is also helpful in the development setting as well, for the same reasons.

PMD is a rich new tool that is growing. With functionalities like these being added to it, it proves to be an even more powerful tool in the future. These are just a few steps in that direction for PMD, but there are many more. It will be a tool to watch in the future.

Bibliography

1. Peter Amey, Praxis High Integrity Systems, "Yours faithfully: an everyday story of formality," Invited keynote address, in "Practical Elements of Safety", Proceedings of the Twelfth Safety-critical Systems Symposium, Birmingham, UK, February 2004. Copyright Spinger-Verlag 2004.

2. W Amme, N Dalton, J von Ronne, and M Franz, "SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form," PLDI 2001.

3. Fred Brooks, "No silver bullet: Essence and accidents of software engineering," IEEE Computer, 20(4):10-19, April 1987.

4. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe, "Extended static checking," Technical Report #159, Compaq Systems Research Center, Palo Alto, CA, December 1998.

5. PMD, `pmd.sourceforge.net`, 2002-2006.

Appendix

```
import java.util.Vector;

class MyClass {
   Vector v;
   Integer a;
   Integer c;
   class Test {
      Integer d;
   }
   class Test2 extends Test {
      Integer f;
   }
   void testme(Test2 one, Test two) {
      System.err.println(one.toString() + two.toString());
   }
   String getInfo() {
      a = new Integer(5);
      return a.toString();
   }
   String getData() {
      return a.toString();
   }
   int main() {
      for (int i = 1; i < 25; i++) {
         int b = 5;
         float c = 56;
         b = b + 34 + (int) c;
         Test2 alpha = new Test2();
         Test2 beta = new Test2();
         testme(alpha, beta);
      }
      return 1;
}  }
```

Figure 4: Another sample test file, used to initially test typechecking and simplerulefile rule sets.

```
jrhott% pmd.sh test.java text basic,imports,unusedcode,controversial
test.java:3      Each class should declare at least one constructor
test.java:4      Use explicit scoping instead of the default package private level
test.java:5      Use explicit scoping instead of the default package private level
test.java:6      Use explicit scoping instead of the default package private level
test.java:7      Each class should declare at least one constructor
test.java:8      Use explicit scoping instead of the default package private level
test.java:10     Each class should declare at least one constructor
test.java:11     Use explicit scoping instead of the default package private level

jrhott% pmd.sh test.java text typechecking
test.java:27     Warning: Casting c (float) as int
test.java:30     Warning: Casting beta (Test2) as Test in function testme

jrhott% pmd.sh test.java text simplerulefile
test.java:16     Warning: Method getInfo in MyClass declared pure but is not
test.java:17     Warning: Variable a changed in method MyClass:getInfo.
```

Figure 5: Sample output running PMD with multiple rulesets on the code in Figure 4 and using the rule file from Figure 3.