2010 2nd International Conference on Software Technology and Engineering(ICSTE)

Static Checking Java with the Java Static Checker

Suzanna Schmeelk
IEEE Member
schmeelk@computer.org

Abstract—**This paper introduces a new open source static analysis tool, the Java Static Checker (JSC). Traditional static analysis tools examine Java code for fault instances considering specific code patterns. JSC is novel in that it examines Java code for potential fault classes based on propagating values and examining fault classes. In this paper, we use the tool to examine programs for potential null pointer dereferences. We empirically evaluate JSC and two other open source tools, Jlint and FindBugs, on a multiple programs. The results show that JSC finds more null pointer dereferences than the other tools.**

Keywords-CASE, Static Analysis, Java

I. INTRODUCTION

Null pointer dereferences are among (if not) the predominate Java programming fault(s) [1], [2]. The null object value has been called the ``billion dollar mistake [3]'' by the inventor as subsequent uncaught null pointer dereferences can cause disastrous system failure. One technique used to locate potential null pointer dereferences is static analysis.

Static analysis is traditional preformed during the software development phase. It can analyzes program syntax and semantics for possible faults. There are multiple proprietary tools available for Java (e.g. Coverity [4], Fortify, Klocwork and XYLEM [5]). Many of the proprietary tools emphasize they locate null pointer dereferences; however, there is a relatively small subset of open source tools available which locate null pointer dereferences and are easy to install. Specifically, FindBugs and Jlint are two easily installed tools which emphasize they locate potential null pointer dereferences. In our experience, current open source tools, designed to lower false positive rates [6], can miss null pointer dereferences. The missed null dereferences led us to build a new open source tool for locating a greater number of potential null dereferences.

II. JSC ANALYSIS METHODOLOGY

JSC is novel in that it is a static analysis tool which uses only standard compiler optimization algorithms for analysis; therefore, it is light-weight Light-weight static analysis requires more effort than using a compiler and less effort than full program verification [7]. Light-weight static analysis tools examine program syntax and semantics for possible faults without proving fault absence.

The light-weight analysis examines programs for fault classes via propagating type values. The tool is comprised of four stages, including (1) parsing the source code and building the abstract syntax tree (via Recoder version 0.85 [7]), (2) generating a control flow graph (CFG), (3) building a static single assignment (SSA) representation and (4) propagating values throughout the program. The tool performs path-sensitive (determines path feasibility), context-sensitive (collects call-site-specific information) and flow-sensitive (examines path traversal) data-flow analysis [12].

JSC uses forward-analyisis to parallelize multiple checkers. A forward-driven methodology computes information relying on information from a given block's predecessors information, as in using reaching-definitions. The current JSC version (1.0) is designed to analyze programs with respect to null pointer dereferences. It does so by propagating object type values throughout a program. The constant propagation of object values is basically an enumerated type where fields consist of the fixed constant type values. As the tool propagates values during constant propagation, it examines object dereferences and reports if a dereferenced object can take on a null value. A benefit of propagating type values is that the values can be used by other parallel checkers without adding additional transformations, a novel design decision.

A. Method One: Genral Analysis

The first example shows a generic class of faults located by the JSC where a class instance variable could cause a null pointer dereference. These cases arise when a class instance variable is neither initialized at declaration nor guaranteed to be initialized in every constructor. The JSC uses data-flow analysis to trace through the constructors and recognize contrasting object values. It subsequently connects the overloaded constructors to a post-constructor join point. The tool is, therefore, not designed to know exactly which constructor is invoked by a client class. This technique is especially useful when client/server code are developed independently (e.g. grid computing environment).

The snippet below belongs to an inner class Node within the BinaryTree.java class. JSC analyzes inner classes first, as inner classes can potentially rely on outer class variables [15]. In the snippet, the object data (of type Node) is not necessarily initialized via the constructors. Subsequently, a call to the toString() method leaving a potential null pointer dereferences on line 45 of the toString() method.

```
44  public String toString(){
45  return data.toString();}
```

B. Method Two: Parameter Analysis

The following example shows a generic class of faults potentially caused from parameters. Following our abstraction objectives, we designed the JSC to analyze object parameters as potentially taking on null values. The selected methodology is based on a client/server paradigm where

978-1-4244-8666-3/10/$26.00 © 2010 IEEE

neither end makes any assumptions about the code. The methodology may be particularly useful for analyzing code including remote procedure calls (e.g. grid services) where client/server code may reside in completely disjoint locations.

In the example below, the JSC locates the potential weakness of the object *input* (of type *BufferedReader*) on line 13. As it is a parameter, it can potentially take on a null value. No matter what assumptions are made by a developer, a safe analysis would recognize the potentially null value.

```
10   public void readLines (Foo input) {...
13       String text = input.readLine( );...}
```

C. Method Three: Phony Assignments

Boolean logic pointer analysis was integrated via phony assignments to guarantee pointers take on their guaranteed conditional (guarded [13], [14]) values. For example, if an object is checked negatively against null (i.e. != null) the semantics guarantee that in the subsequent *then* statement block the object cannot take on a null value. The conditional also guarantees that in the subsequent *else* statement block the object can take on a null value. Phony assignments are inserted into the code to guarantee that guarded objects are given the appropriate values within their respective scope. This methodology eliminates false positives in these cases.

1) General Conditional Cases: The following example shows where the JSC prevents a false positive via the phony assignment mechanisms. JSC does not report potential null pointer dereferences for the BinaryTree instance objects leftTree (on line 71). As described, our phony assignment mechanism guarantees that semantic analysis is considered when examining pointers within conditional blocks.

```
67   public BinaryTree(BinaryTree rt,...){
70   if (rt != null) {
71   root.right = rt.root;}
```

2) Complex Conditional Cases: The following two examples show the cases where conditional semantics are appropriately analyzed. In the first example, the JSC does not report a potential null pointer dereference on root in the second conditional expression on line 92. Root is a class instance variable that is neither initialized at declaration nor guaranteed to be initialized in every overloaded constructor. The semantics of Java guarantee a left-to-right evaluation within a conditional statement [15]. A phony assignment is inserted between conditional expressions to guarantee that the appropriate value is further considered during analysis. In this case, JSC does not generate a false positive report on the object *root.left*.

```
90   public BinaryTree getLS(){
91     if (root != null
92         && root.left != null) {
93           return new BinaryTree(root.left);}
```

3) Assert Cases: In the example below, the JSC does not report a null pointer dereference of the PrintStream object output on line 29. Although the parameter object output can potentially take on a null value, the assert statement on line 24 guarantees the object will subsequently be non-null. JSC inserts a phony assignment to incorporate the appropriate semantics of assert statements. (Note, JSC does not report a fault for the Hashtable object dict on line 25, as it is a class instance variable initialized at definition.)

```
23     public void generateOutput (Foo out){
24       assert(out != null); ...
29       out.print(word + ":");...}
```

D. Method Four: Limiting Output Redundancy

Limiting Reports per Line: In the following example, JSC reports a potential null pointer dereference on the object root on line 105. Root is a class instance object neither initialized at declaration nor guaranteed to be initialized in every constructor. The following example is important, as it shows a case where the JSC is designed to report strictly one fault per line per checker. Therefore, in the following example, only one fault will be reported instead of two faults. This design arose from extensive work with a proprietary static analysis tool where a single fault could be reported multiple times; therefore, aggravating developers. We chose to design a tool with limited repetitive output. If developers fail to fix all potential faults of the same class within a line, subsequent analysis will report the uncorrected potential fault.

```
104     public boolean isLeaf() {
105      return (root.left == null
106      && root.right == null);}
```

1) Limiting Subsequent Faults: At least four potential weaknesses appear in the following code snippet for the object StringBuffer sb. As sb can potentially take on a null value, the dereferenceS on lines 135, 138 and 141 are unsafe. JSC can limit the output to report strictly the first dereference along distinct paths; therefore, solely reporting a dereferences on line 135. Our phony assignment mechanisms guarantee that node dereferences on line 141 is not reported by JSC.

```
132    void preOrderT(StringBuffer sb) {
134      for (int i = 1; i < depth; i++) {
135      sb.append("  ");}
137      if (node == null) {
138        sb.append("null\n");}
140      else {
141        sb.append(node.toString());...
```

E. Method Five: Aliase Analysis

Alias analysis is an open research area. Aliases arise from either a direct object assignment or a call to a method which returns an object pointer.

Both cases are being considered as JSC is tuned. In the example below, if the *bR* object dereference were strictly from the call to the object's *readLine()* method, then *data* (of type *String* on line 159) would also take on a potential null value. However, as seen in the snippet a call to the *readline()* method is subsequently used for call to the returned object's *trim()* method returns a non-null value guaranteeing *data* to not take on a null value.

```
157     BT readBT(BufferedReader bR){
158     String data = bR.readLine().trim();...
159     BT left = readBinaryTree(bR);...}
```

III. Empirical Evaluation

Thirteen student Suduko programs consisting of 171 classes were used to evaluate JSC (v.1.0). As can be seen in Table I, JSC located 8 faults. All false positives were eliminated using the methods described in Section II.

TABLE I. JSC Warning Output

Case	JSC
1	2
2	0
3	0
4	0
5	0
6	0
7	2
8	0
9	0
10	4
11	0
12	0
13	0
Total	8

The first four faults found by JSC, consisting of the faults in Case 1 and two in Case 7, are a result of uninitialized class objects that are dereferenced in different class methods. The second four faults consisting of the faults in Case 10 are a result of a null parameter. In all cases the parameter is dereferenced without a guard [15]].

The last four faults, consisting of the faults in Case 10, are a result of null parameters. In the four cases, the parameter is dereferenced without a guard. In the first two faults, the object is dereferenced for a class variable; and, in the second two faults, the object is dereference with a call to a *size()* method.

IV. Alternative Static Analysis Methods

A. FindBugs

FindBugs [16], [17] is a pattern-driven tool for examining Java byte-code and source-code. FindBugs was initially developed by William Pugh at the University of Maryland. The architecture relies on input bug patterns, or rule sets, to locate instances of faults. Bug patterns are ``code idioms that are likely to be errors [17]." It currently has the following: over 300 pattern matchers, rule sets written in Java and a plug-in architecture. Cole, et al. [17], emphasize that it finds one fault per 0.8 - 2.4 KLOC on non-commercial source code.

FindBugs is designed to keep analysis simple and not to generate too many warnings. It provides a framework for doing both forward and backward data flow analysis, including intraprocedural and type analysis. The FindBugs literature states that the tool reduces false positives via a JDK library context-sensitive analysis. It does not perform interprocedural context-sensitive analysis; but, can use annotations and programming behavior to improve parameter analysis [18], [19].

FindBugs examines code for potential null pointer dereferences using a forward driven analysis methodology to compute static single assignment (SSA) form; it, then, uses backward-driven analysis to detect guaranteed dereferences. Hovemeyer et al. [6] state that dereference detection is optimized not to include dereferences where semantics guarantee an object to be non-null. They also state that null pointer dereferences include references to pointers that have a ``high likelihood of causing a null pointer exception" such as passing a value to a parameter that must be non-null. The object value propagated during analysis falls within the set *Null, Null-E, NonNull, Checked NonNull, No Kaboom NonNull, NSP, NSP-E, NCP* where the values are associated with object values along the control flow of the program [19]. In the set, the values *Null* and *NonNull* are guaranteed values taken on by objects; whereas, the remaining values depend on exception, complex and infeasible paths. The values propagated by FindBugs are solely used for checking null pointer dereferences. FindBugs reports both the location where an object value is known to be null and a pruned set of locations that dereference the potentially null object. At the current time, the tool does not consider aliases.

B. Jlint

Jlint [20], designed by Konstantin Knizhnik and Cyrille Artho, is another static analysis tool. The tool has two components: a syntax checker, AntiC, and a semantic verifier. Jlint's semantic verifier is designed to look for three categories of faults: synchronization, inheritance and data flow. The semantic checker examines data flow at two levels, locally and globally. A local data flow analysis examines a set of statements within a given scope where there is no control flow transfer [8]. A global data flow analysis, also known as intraprocedural analysis, examines statements inclusive of a method [8]. At the two levels, with respect to data flow analysis, Jlint calculates ranges of values for expressions and local variables and saves the calculations into a variable descriptor. A handy feature of Jlint is the implementation of a *history* file, which can be stored to suppress warnings during subsequent analysis.

C. Analysis Methodology Summary

A summary of the data flow analysis methodologies used by the tools is described in Table II. The first row lists the discussed tools. The first column presents their analysis methodologies. The code row describes the type of code that is analyzed-either source code or binary code. The type row presents the overall methodology of the tool--either pattern matching, syntax/semantic or class. Tools either perform a forward or backward analysis, depicted in the direction row. The data flow analysis of procedures, in the row procedural, is either intra-procedural or inter-procedural. Alias analysis, shown in the alias row, tracks aliases that reference the same location in memory. Data flow sensitivities, described in the sensitivities row, include path, context and flow.

TABLE II. TOOL ANAALYSIS METHODOLOGIES

Methods	FindBugs	Jlint	JSC
Code	S, B	B	S
Type	Pattern	Syntax, Semantic	Class
Direction	F, B	F	F
Alias	-	-	-
Sensitivites	Path, Flow, Context	Path, Flow	Path, Flow, Context

TABLE III. TOOL WARNING OUTPUT

Case	Jlint	FindBugs	JSC
1	0	0	2
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	1	2
8	0	0	0
9	0	0	0
10	0	0	4
11	0	0	0
12	0	0	0
13	0	0	0
Total	0	1	8

D. Complete Empirical Evaluation

Thirteen student programs consisting of 171 classes were used to evaluate JSC (v.1.0). As can be seen in Table III, JSC located 8 faults. The fault located by both FindBugs and JSC in Case 7, was the result of a dereferenced uninitialized class instance object.

V. CONCLUSIONS

In this paper we presented and empirically evaluated our light-weight static analysis tool designed to locate classes of faults rather than instances of faults used by popular pattern matchers. The empirical study showed that examining a program for fault classes can lead to discovering more faults than examining code for fault instances without increasing false positives.

Future work on the JSC includes expanding and parallelizing the checking capabilities of our tool to include out-of-bound indexing, concurrency issues, type checking and resource problems ranging from hanging open sockets to hanging open databases. Additionally, we are tuning the JSC to incorporate user defined assert statements, integrated database and including a more comprehensive alias analysis.

REFERENCES

[1] S. Schmeelk, B. Mills, and R. Noonan, "Managing post-development fault removal," in Information Technology: New Generations, 2009, ITNG'09. Sixth International Converence on, April 2009, pp. 205-210.

[2] Coverity, "The next geneartion of static analysis," 2007. [Online]. Available: hhtp://www.coverity.com/whitepapers.

[3] T. Hoare, "Null references: The billion dollar mistake." [Online]. Available: http;//qconlondon.com/london2009/speaker/Tony+Hoare

[4] D. Engler, D.Y.Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a genearl approach to inferring errors in systems code," SIGOPS Oper. Syst. Rev., vol. 35, no. 5, pp.57-72, 2001.

[5] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java, " in ICSE'09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering. Washington, DC, USA: IEEE Computer Society, 2009, pp. 133-143.

[6] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many, " in PASTE'07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. New York, NY, USA: ACM, 2007, pp. 9-14.

[7] D. Evans and D. Larochelle, "Improving security using extensivle lightweight static analysis, " Software, IEE, vol. 19, no. 1, pp.42-51, jan/feb 2002.

[8] U. Khedker, A. Sanyal, and B. Karkare, Data Flow Analysis: Theory and Practice. Boca Raton, FL, USA: CRC Press, Inc., 2009.

[9] K. Miller, L. Morell, R. Noonan, S. Park, D. Nicol, B. Murrill, and M. Voas, "Estimating the probability of failure when testing reveals no failures, "Software Engineering, IEEE Transatcations on, vol. 18, no. 1, pp.33-43, Jan 1992.

[10] B. Lisokov and J. Guttag, Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[11] A, Ludwig, "Recoer," 2008. [Online]. Available: http://apps.sourceforge.net/.

[12] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," Commun. ACM, vol. 53, no. 2, pp.66-75, 2010.

[13] E. W. Dijkstra and W. H. Fiejen, A Method of Programming. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988.

[14] E. W. Dijkstra, A Discipline of Programming. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997..

[15] B. Eckel, Thinking in Java. Prentice Hall PTR, 1998..

[16] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warkings on producation software, " in PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT

workshop on Program analysis for software tools and engineering . New York, NY, USA: ACM, 2007, pp. 1-8.

[17] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens, "Improving your software using static analysis to find bugs, " in OOPSLA'06 Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. New York, NY, USA: ACM, 2006, pp.673-674..

[18] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs, " IEEE Software, vol. 25, no. 5, pp.22-29, 2008..

[19] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a stiat analysis tool to find null pointer bugs," in PASTE'05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. New York, NY, USA: ACM: 2005, pp. 13-19.

[20] K. Knizhnik and C. Artho, "Jlint manual," [Online]. http://jlint.sourceforge.net/..