Software Verification Tools Overview

Clément Beffa, Vincent Pazeller & Olivier Gobet
Swiss Federal Institute of Technology
{firstname.lastname}@epfl.ch

Abstract

Bugs are becoming a bigger concern nowadays as we are seeing their huge cost. Academics are building numerous tools to get rid of them with more or less success. In this paper, we make an overview of bug finding tools and focus deeper on those targeting Java code. We explain the purpose of each of them and test automatic tools on a specially built Java test case in order to see their accuracy. Finally, we develop our vision of a theoretical meta-tool which would be able to combine the best of them.

1. Introduction

Software Analysis and Verification is a growing open research area, because software is becoming omnipresent and bugs start to cost a fortune. As they were estimated to cost the U.S. Economy $59.5 Billion Annually[1]. Software errors can be as costly as the crash of the Ariane rocket in 1996 which was estimated to a loss of US$370 million[2] or, even worst, it could involve human deaths.

Nowadays, many academics and open source projects are building tools to get rid of as many bugs as possible. They are using many different techniques and each tool serves different purposes. Our study tries to have an overview of what is available and how it can serve us to improve software quality.

2. Overview of Tools

There are many available tools on the internet. We looked at 37 non-commercial and downloadable tools. A list of them and their characteristics is available in the appendix. We found that there is huge disparity among them. For example, some tools like ARMC are alpha versions, coming from a publication concept, whereas tools like FindBugs are stable versions, used by tens or hundreds of thousands of people and are sponsored by big corporations like Google and Sun Microsystems. Each tool tries to find some set of bugs, but does not necessarily effectively found them as advertised. Bugs that can be detected are, for instance, null-pointer dereferences, divisions by zero, inconsistencies, deadlocks, memory leaks, security flaws or array out of bounds. As the figure 1 shows, most tools try to focus on popular languages like C,C++ and Java, but some other, like armc or pale, apply to homemade languages, for the purpose of proving validity of personal projects, thus making them unusable on real project.

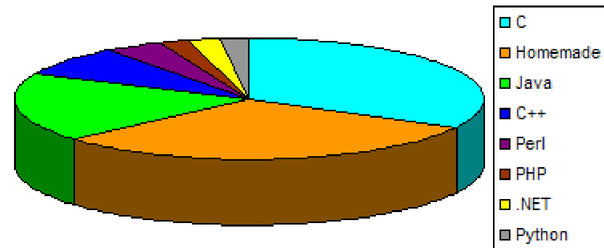

Figure 1. Languages repartition

3. Details of Tools Supporting Java

3.1. Jahob

Jahob is a verification system that supports a subset of Java. It is not designed to automatically analyze production code but only to prove dynamically allocated data structures and arrays. Thus, it requires the Java code to be annotated with pre-conditions, invariants and post-conditions as shown in the code example below.

```
private static void
  checkingWithdraw(int amount)
/*:
requires "amount > 0 &
  checkingBalance >= amount"
modifies checkingBalance
ensures "checkingBalance =
  old checkingBalance - amount"
*/
{
```

```
  checkingBalance =
    checkingBalance - amount;
}
```

There is no GUI and the CVC package is required. The output is easy to read and allows unproved methods to be pinpointed.

3.2. F-Rex

F-Rex is composed of two tools : Jreg and Jfree. Its purpose is to provide compile-time memory management by analyzing lifetime of objects and deallocation in order to verify that program are memory safe. As we can see in the example below, the output is very detailed thus making it unpractical to analyze large project.

```
.method public static
  main([Ljava/lang/String;)V
      .limit stack 1
      .limit locals 1
      create_ru 1 ;o3062:[]:r2b |
      new_in_r 1 A
      astore_0
      push_r 0
      aload_0
      ; jreg added-> to attach
      ; attribute at call opcode
 label1:
      invokespecial A/<init>()V
      aload_0
      astore_0
      push_r 1
...
```

3.3. Daikon

Daikon is a dynamic detector of likely invariants. It does not detect bugs directly but helps to have a better understanding of the program which could help to find bugs. In addition to support Java, it also supports C, C++ and Perl. The most useful functionality of Daikon is that it allows to automatically annotate programs with the found invariants, which could be very helpful when checking bugs with tools that require annotation like ESC/Java for example.

3.4. Purity Analysis Kit

The purity analysis kit allows to check for purity of Java methods. Pure method is defined as method that does not mutate any object that existed before the method was invoked (i.e. border effects). As shown in output sample, every method is analyzed and labeled as PURE or not and why. Such tools do not find bugs but they can help to understand the program and by knowing that a method is pure it proves that this method do not interfere with other computations.

```
void swapValRight(Value n)
NOT HEAP PURE
 "this": mutation on this.(right|value)
 "n": mutation on n.(value|right)
```

3.5. ESCJava2

ESC/Java2 is a static analyzer of the program code with formal annotations. It consists of parsing, type and static checking. It can be used directly on production code but produces many warnings in this case, as it reasons about each methods individually. Thus it is better to annotate programs to reduce the number of warning as shown below.

```
class A{
 byte[] b;
 //@ ensures b != null && b.length = 20;
 public void n() { a = new byte[20]; }
 public void m() { n();
   b[0] = 2;
 }
}
```

This tool is platform independent and has a GUI which unfortunately is not that intuitive. The main issue is that ESC/Java2 only supports Java 1.4 which restraints his usability.

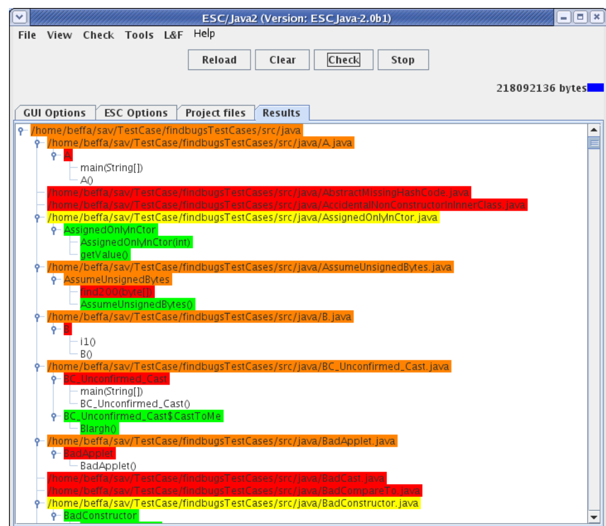

Figure 2. ESCJava2 GUI, colorful but not so intuitive

3.6. Findbugs

Findbugs is a bug finder tool based on static analysis of Java bytecode, looking for occurrences of bug patterns. Thus, it only requires compiled class files to work but allows to specify source files in order to see directly the related suspicious code. It has a command line interface and a good graphical user interface (see figure 3) which groups bugs by categories and allows to browse buggy source code.

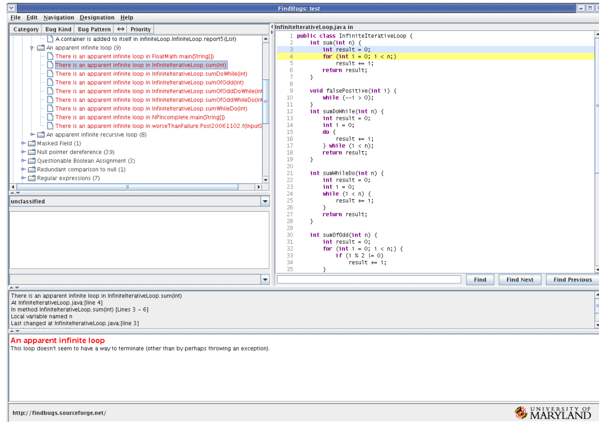

Figure 3. FindBugs GUI, groups bug together by categories

The bugs are divided into three categories:

- Correctness bug : Those which are probable bug
- Bad Practice: Those which violate recommended coding practice
- Dodgy: Those which are confusing and prone to errors

Findbugs tries to avoid false positive as much as possible, especially on correctness bugs. Findbugs is also very flexible as it has a plug-in architecture which allows anyone to add new bug detectors.

3.7. JLint

JLint is a full automated Java verification tool. It takes Java bytecode (.class) as input and prints error/warning messages accordingly. It discovers several bugs notably by performing data flow analysis, like for instance Nullpointer exceptions, arithmetic exceptions, array out of bounds or deadlocks. It is also able to detect some bad programming habits like variable shadowing, zero operands or weak comparisons. Unfortunately, JLint does not provide any GUI. Nevertheless, its shell commands are user friendly, leading in a comfortable usage.

3.8. PMD

PMD includes many detectors for bugs that depends on programming syntax. PMD scans Java source code and looks for potential problems like empty statements, unused local variables or duplicated code. It offers the possibility to suppress warnings for classes, methods or lines by annotating the source code. The analysis is made using abstract syntax tree built on the source code by a JavaCC generated parser.

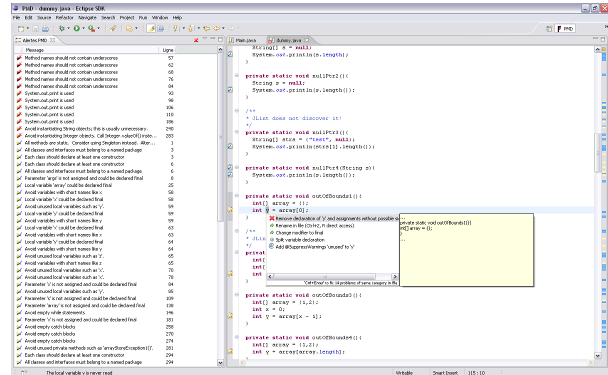

Figure 4. PMD GUI as Eclipse plugin

PMD could be used via command line or integrated into IDEs like JDeveloper, Eclipse, JEdit, JBuilder or Emacs. PMD is based on rulesets which are provided or could also be home made. The creation of rulesets could be done in two ways: either write a rule using Java or write an XPath expression. The recommanded rulesets are unusedcode.xml, basic.xml, import.xml, favorites.xml.

Moreover, PMD's Copy/Paste Detector is an additional tool, used at command line or with a GUI, to find duplicated code in languages such as Java, JSP, C, C++, and PHP. There is also another GUI tool, PMD Rule Designer, that allows to generate AST's and to test out XPath expressions.

3.9. jCUTE

jCUTE for java (Concolic Unit Testing Engine) is a tool designed to systematically and automatically test concurrent Java programs. The algorithm in CUTE uses a technique called concolic execution whose strategy is to explore all distinct execution paths of a program with data inputs. It allows random testing and smart fuzz testing in addition to concolic testing.

It can automatically generate optimal JUnit test cases for sequential Java programs and optimal number of test inputs for path coverage, branch coverage, erroneous execution. About concurrent Java programs, jCute can catch actual data-races and deadlocks without any false warning.

Moreover it can be used directly in the source code by writing assertions. jCUTE can either be executed at command line or with a GUI.

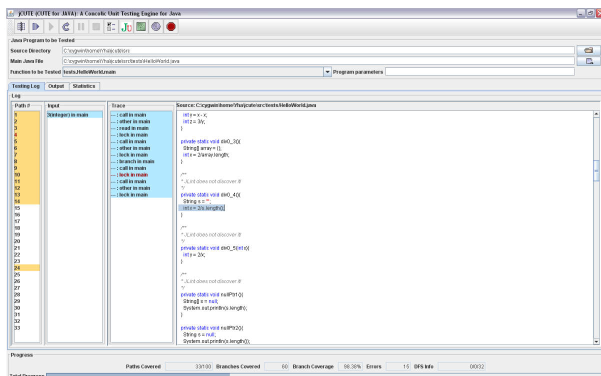

Figure 5. jCUTE gui

As jCUTE actually runs the java program, it catches java runtime exceptions in order to detect bugs. For instance, it discovers infinite recursions by StackOverFlow, infinite loops by OutOfMemory. However we found a special case, when it tests $while(true)$ statement, it loops infinitly without giving any errors or throwing any exceptions. This is due to the fact that neither the stack nor the memory are used,thus, no exception is thrown.

A big drawback of jCUTE is that it does actually run the potentially bugged code. Therefore, if a malicious bug is writing on the hard drive (deleting or modifying files), the bug we wanted to avoid is released by the verification tool and can lead in the loss of data, potentially followed by other injuries. As a result, jCute seams to be unsafe!

4. Experiment on test case

We tested automatic tools on a homemade test case containing 32 bugs. There were different ways to achieve the same bug type, like for example five kind of division by zero. As shown in figure 6, tools do not found the same bugs and do not always found all bugs in the same category. We can see that jCUTE gets the best mark but it is a bit unfair to compare to the others as it does found error on runtime and not by static analysis. The dash in jCUTE is for the $while(true)$ loop as it does not find the bug but the tool effectively loops for ever. The dash in ESC/Java2 means the bug could not be tested because ESC/Java2 supports only Java 1.4.

5. Theorical meta-tool

As it was mentioned in [3], creating a meta-tool that regroups automatic bug-finders is a good solution to provide powerful bug detection. This is possible since each tool uses different approaches and, therefore, detects different bugs. Some studies have even showed that different tools using similar approaches do not completely overlap each other. The major problem with meta-tools is that they should not blindly forward errors provided by underlying verification tools. Otherwise, each incorporated tool will potentially generate the warning for one original bug. This leads in many duplicated errors, thus flooding the user with thousands of errors. Such a system is clearly not usable.

In order to avoid repeating warnings, some groups propose to filter the output of underlying tools. Unfortunately, this is far from trivial since each tool uses different conventions and this leads in filtering inaccuracies. We propose here to standardize verification tools warnings. For instance, before incorporating a specific tool to the meta-tool, one should either modify its warnings system (when sources are available) or write a wrapper. It would then be easier to detect similar warnings among tools, thus allowing a more accurate filtering. Moreover, this approach allows the meta-tool to determine the category and the severity of a bug. This could then enable an interesting feature: it would be possible to the user to specify the kind and/or the severity of researched bugs. This is very important to programmers who tends to debug their program in a bottom-up[1] fashion. Ranking warnings also allows the design of new strategies, as described below.

The idea is to organize the meta-tool in layers. In fact, tools detect different kinds of bugs. For instance, PMD concentrates on syntax patterns while JLint performs intra-procedural data flow analysis. As mentioned before, developers usually begin to search syntax mistakes, continue with intra-procedural bugs, then, inter-procedural bugs and finally check for global-scope mistakes. Therefore, at first, the meta-tool would run a selection of tools specialized in syntax analysis. Note that those tools may not only provide syntax analysis. In that case, since errors have been standardized (see above), the meta-tool should be able to mask errors not related to the current layer. PMD seems to be a good candidate to be part of the syntax layer since it focuses on syntax patterns. Note that in order to get an efficient meta-tool, more than one tool should be selected for each layer. For the intra-procedural layer, JLint could be incorporated. Unfortunately, there are currently not many tools providing inter-procedural and global-scope bugs detection. Related problems seem to be harder to solve.

Another advantage of the meta-tool approach is its maintenance. As verification theory advances, new tools are deployed and lots of work must be redone (to incorporate older techniques). If the designers know that the new tool will be incorporated in a meta-tool, older techniques do not need to be re-implemented since they are likely included in tools

[1]From the most specific to the most general.

Type	DIV 0					Null ptr				Bounds					Inf L				Inf R		
Bug	1	2	3	4	5	1	2	3	4	1	2	3	4	5	1	2	3	4	1	2	3
JLint	x	x	x				x		x	x		x	x	x				x			
jCUTE	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-	x	x	x	x	x	x
ESC / Java 2	x	x	x	x	x	x	x	x	x	x		x	x								
FindBugs						x	x			x		x	x					x			
PMD																					

Type	Type Ov		Cast	String Cmp		Stream Close		Store	Deadlock		Score
Bug	1	2	1	1	2	1	2	1	1	2	
JLint											10 / 31
jCUTE			x							x	23 / 31
Esc / Java 2	-	-	x	-	-	-	-	x			14 / 31
FindBugs						x	x				8 / 31
PMD											0 / 31

Figure 6. Experiment results

already available in the meta-tool.

Finally a GUI seems to provide an efficient usage. A GUI has the advantage of providing a general view of the tested program by regrouping bugs in categories and proposing links from bugs to source code. With a GUI it is also more practical to customize the soft via options.

6. Conclusion

During our study, we have found that many different tools exist, used in different contexts, for different kinds of bugs. Each of them is designed and have to be used i its own way, there does not exist a standard or a main one. After scratching a bit deeper, we found that most bug-finders are unsound. By example finding null pointers could be really easy when it is limited to a function scope, indeed an intra-procedural analysis is sufficient, but when it is due to side effects it will require an inter-procedural analysis which is much more complex and not usual in bug finding tools.

By the serial of tests that we elaborated, we also noticed that the creation of bugs is not as simple as we could think. Writing pertinent bugs could be somehow harder than doing a bug-free code. Moreover, installing and using bug finding tools on our test case is not as intuitive as it should be.

By our own analysis we confirm what was concluded in [3]. That is, a meta-tool which regroups different bug-finders is a more secure and complete way to detect bugs. Therefore we defined how it could be possible to implement such a meta-tool without entombing the user under a huge heap of warnings and errors.

As we saw in the tools list, many more tools would be interesting to test deeper. Focusing on the C++ bug tools would be challenging or to look at new upcoming tools like Chord. Software analysis and verification research are still going strong and bugs are probably not on an extinction way.

References

[1] Software Errors Cost, `http://www.nist.gov/public_affairs/releases/n02-10.htm`

[2] Ariane 5 Flight 501, `http://en.wikipedia.org/wiki/Ariane_5_Flight_501`

[3] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java *15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245-256, 2004.

Nom	version	last version date	languages	os	dependency	requirement	type of bugs	description / remarque
Jahob system	preview	04 / 08 / 2007	subset of Java	Linux, Mac, Win	Ocaml, CVC3	annotated code in Isabelle syntax	- high-level data structure consistency	- static analysis and verification system for modular analysis of imperative computer
FindBugs	1.2.1	04 / 25 / 2007	Java	Linux, Mac, Win	Java 1.4.0 or above	at least 512 MB of memory	- Correctness bug - Bad Practice - Dodgy	- static analysis to look for bugs in Java code
PMD	3.9	12 / 19 / 2006	Java	Linux, Mac, Win	Java	java source code	-Possible bugs -Dead code -Suboptimal code -Overcomplicated expressions -Duplicate code	- PMD scans Java source code and looks for potential problems
F-Rex	1.1.0	04 / 15 / 2007	Java	Linux, Mac, Win	Java, Soot, Kaffe, JikesRVM	-	- memory management	- explicit free and region support for Java
Purity Analysis Kit	0.09	07 / 31 / 2006	Java (.class)	Linux	JDK 1.5	???	- detects pure methods à la JML	- pointer and purity analysis tool for Java programs
ESC/Java2	2.0b0	- / 10 / 2006	Java	Any	Java	Java JML-annotated program	- common run-time errors	- static analyzer
Daikon	4.2.16	01 / 05 / 2007	C, C++, Java, Perl, IOA	Any	Java 5.0 or above, JVM	???	- invariant detector	- easy to extend to other applications (e.g., an interface exists to the Java PathFinder model checker) - front end for Java (Chicory), for C/C++(Kvasir, mangel-wurzel)
CUTE	1.0.1	29 / 06 / 2006	C, Java	Linux, Win	Java 1.4 or above, bash shell (optional) optional: Win: Cygwin	???	- concolic execution	- CUTE (a Concolic Unit Testing Engine for C and Java) is a tool to systematically and automatically test sequential C programs (including pointers) and concurrent Java programs. - Automatic testing of C and Java programs, there is no need to write test cases, test cases are generated by dynamic analysis of source code.
Jlint	3.0	21 / 06 / 2004	Java	Linux, Win	Java	???	- bugs - inconsistencies - synchronization problems	- check your Java code and find bugs, inconsistencies and synchronization problems by doing data flow analysis and building the lock graph.
KeY	1.0.0	- / 04 / 2007	Java Card	Linux, Solaris, Win	JDK1.4.1	ocl/uml, jml	- invariants unconsistency	- static analyzer
Comfort	2.0	02 / 01 / 2006	Construction and Composition Language (CCL)	Any	Copper	CCL	Assertion-based: - Counterexample-Guided Abstraction - State/Event-based Software Model Checking - SAT-based Predicate Abstraction - Automated Assume-Guarantee Reasoning - Compositional Deadlock Detection - Software Certification - Component Substitutability	- the Component Formal Reasoning Technology (ComFoRT) is a reasoning framework for predicting whether a system will satisfy its safety, reliability, and security requirements. In ComFoRT, these requirements are encoded as behavioral assertions that are verified automatically.
Berkeley Lazy Astraction Software Verification Tool	2.0	- / 12 / 2005	C	Linux, Win	Ocaml, cygwin, Symplify Theorem Prover	annotated preprocessed C code	- safety	- safety assertion check of C programs
ARMC	1.0 ?	04 / 12 / 2007	armc	Linux	/	???	- verification of reachability - termination properties	- it is a tool for the verification of reachability and termination properties
Hob system	0.1.0	12 / 09 / 2005	homemade	Linux	Ocaml	write code for abstraction, implementation and specification	- data structure consistency	- verify sophisticated properties of programs that manipulate complex, heterogenous data structures
Leak contradictor	1.0	- / 11 / 2006	C	Any	Java 1.5, Apache Ant, Crystal Framework	C preprocessed code	- memory leaks	- memory leak detection tool for C programs
TVLA	2(alpha)	- / 08 / 2004	tvp	Any	Java 1.4.2, Graphviz	tvp file	- data Structures	- properties checker of heap allocated data
PALE	1.0-9	03 / 08 / 2007	pale	Linux, Win	MONA	annotated program	- null-pointer dereferences - memory leaks - violations of assertions - graph type errors	- expressing assertions about the heap structure of imperative languages
UCLID	1.0	06 / 02 / 2003	ucl	Linux	???	writing model	- model checking - correspondence checking - deductive verification - predicate abstraction-based verification	- verification tool for Infinite-State Systems

Spec#	1.0.7301	- / 05 / 2007	spec#	Win	-	spec# program	- invariants unconsistency	- static verifier
KIV	???	06 / 04 / 1996	-	Any	-	making your own code ?!?!	- formal specification and verification	- The KIV system, as a tool for interactive proof engineering, has turned out to be very successful in verification tasks which cannot be tackled fully automatically.
Spin	4.2	01 / 05 / 2007	C	Any	gcc, Tcl/Tk Wish optional: Yacc, Dot, JSpin, Ltl2Ba	???	- search algorithm - verification option - complete language	- in April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM.
CBMB	2.5	- / 08 / 2006	ANSI-C	Linux, Win	Win: CL	C	- bounds checking - dynamic memory allocation - ...	- uses Bounded Module Checking
MAGIC	1.0	05 / 06 / 2004	C	Linux, Win	Win: Cygwin	FSP notation to specify state machines	- verify that an implementation conforms to its specification	- implementations could be concurrent - command line base tool
Saturn	1.0	- / - / 2004	C	Linux	-	-	- SAT-based	- the goal of the Saturn project is to statically and automatically verify properties of large (meaning multi-million line) software systems. - the release includes a sound alias analysis and an unsound (bug-finding) null dereference analysis for C programs
CAsCaDE	1.0 ?	- / 01 / 2006	C	Linux	CVC Lite, EDG	control file specifies the assertion(s) to be checked (XML format)	???	- tool to check assertions in C programs as part of multi-stage verification strategy - it takes as input a C program and a control file
SatAbs	1.8	- / - / 2005	ANSI-C	Linux, Win	Win: CL (comes with Microsoft Visual Studio) or Visual C++ Express Linux: gcc/g++	Model Checker for SATABS (recommend either Cadence SMV or BOPPO)	- verifying array bounds (buffer overflows) - pointer safety - exceptions - user-specified assertions	- it allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions - furthermore, it can check ANSI-C for consistency with other languages, such as Verilog. SATABS computes an abstraction of the program in order to handle large amounts of code. - eclipse plugin
ObjectCheck	???	- / 05 / 2006	code OO via xUML	???	xUML COSPAN OOA	construct xUML schema	- verification of liveness and safety specification of xUML system	- the key approach of the project is to couple design techniques with system verification. The design techniques propose software systems as executable object-oriented models specified using high-level design description languages (e.g., UML statecharts, etc.). Verification is conducted by direct application of model checking-based analysis to program designs. - PROGRAM NOT FOUND
B-toolkit	5.7	- / 02 / 2002		Win	-	-		- it provides full support the B-Method, is a mature, integrated suite of tools, built partly on the B-Tool interpreter, and covering many aspects of software engineering
Code Surfer	2.0	- / - / 2007	C, C++	Linux, Solaris, Sparc, Win	Code Sonar	-	- null-pointer dereference - divide-by-zero - resource leak - buffer overflow - and many others	- code analysis tool
Why	2.0.3	- / 04 / 2007	-	Linux, Win	-	annoted programs(imperative language), OCaml-like syntax	-	- verification conditions generator.generates proof obligations for many systems: the proof assistants Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, Mizar
Calysto	1.7	- / 05 / 2007	-	Linux, Mac	-	-	???	- bug hunting companion rather than a formal verification tool
Flawfinder	1.27	17 / 01 / 2007	C, C++	Linux	Python	???	- security flaws	- program that examines source code and reports possible security weaknesses (``flaws'') sorted by risk level.
Splint	3.1.1	30 / 10 / 2003	C	Linux		???	- security vulnerabilities - coding mistakes	- tool for statically checking C programs for security vulnerabilities and coding mistakes
RATS	2.1	24 / 09 / 2002	C, C++, Perl, PHP and Python	Linux, Win	Expat	???	- buffer overflows - TOCTOU (Time Of Check, Time Of Use) race conditions	- RATS scanning tool provides a security analyst with a list of potential trouble spots on which to focus, along with describing the problem, and potentially suggest remedies
FxCop	1.35	06 / 23 / 2006	.NET	Win	.NET 2.0	???	- Defects in Library design - Localization - Naming conventions - Performance - Security	- code analysis tool that checks .NET managed code assemblies for conformance to the Microsoft .NET Framework Design Guidelines.
BOON	1.0	03 / 07 / 2002	C	Linux		???	- buffer overrun	- automatically finding buffer overrun vulnerabilities in C source code
C Code Analyzer - CCA	0.8	30 / 04 / 2005	C	Linux	Ocaml, Perl	???	- memory leak detection - multiple/dangling free detection - array out of bound accesses - potential bufferoverflow detection	- static analysis tool for detecting potential security problems in C source code.

```java
/* Bugs Test Case */
import java.io.FileInputStream;

class A{}


class TestSuite{

  public static void main(String[] args){

    divO_1();
    divO_2();
    divO_3();
    divO_4();
    divO_5(0);

    nullPtr1();
    nullPtr2();
    nullPtr3();
    nullPtr4(null);

    outOfBounds1();
    outOfBounds2();
    outOfBounds3();
    outOfBounds4();
    int[] array = {};
    outOfBounds5(array);

    infiniteLoop1();
    infiniteLoop2();
    infiniteLoop3();
    infiniteLoop4();

    infiniteRecursion1();
    infiniteRecursion2(2);
    infiniteRecursion3();


    typeOverFlow1();
    typeOverFlow2();

    InvalidCast1();

    BadStringCmp1();
    BadStringCmp2();

    BadStreamClose1();
    BadStreamClose2();

    DeadLock1.startThread();
    DeadLock2.startThread();
  }

  /***********
  ****BUGS****
  ************/

  private static void divO_1(){
    int x = 0;
    int y = 3/x;
  }

  private static void divO_2(){
    int x = 1;
    int y = x - x;
    int z = 3/y;
  }

  private static void divO_3(){
    String[] array = {};
    int x = 2/array.length;
```

```
  }

  private static void div0_4(){
    String s = "";
    int x = 2/s.length();
  }

  private static void div0_5(int x){
    int y = 2/x;
  }

  private static void nullPtr1(){
    String[] s = null;
    System.out.println(s.length);
  }

  private static void nullPtr2(){
    String s = null;
    System.out.println(s.length());
  }

  private static void nullPtr3(){
    String[] strs = {"test", null};
    System.out.println(strs[1].length());
  }

  private static void nullPtr4(String s){
    System.out.println(s.length());
  }

  private static void outOfBounds1(){
    int[] array = {};
    int y = array[0];
  }

  private static void outOfBounds2(){
    int[] array = {1,2};
    int[] ids = {-1};
    int y = array[ids[0]];
  }

  private static void outOfBounds3(){
    int[] array = {1,2};
    int x = 0;
    int y = array[x - 1];
  }

  private static void outOfBounds4(){
    int[] array = {1,2};
    int y = array[array.length];
  }

  private static void outOfBounds5(int[] array){
    int y = array[array.length-1];
  }

  private static void infiniteLoop1(){
    while(true){}
  }

  private static void infiniteLoop2(){
    for(int i = 0 ; i < 1 ; i--){}
  }

  private static void infiniteLoop3(){
    int x = 1;
    for(int i = 0 ; i < x ; i++){
      x++;
    }
  }
```

```
private static void infiniteLoop4(){
  int x = 2;
  for(int i = 0 ; i < x ; ){}
}

private static void infiniteRecursion1(){
  infiniteRecursion1();
}

private static void infiniteRecursion2(int x){
  if(x > 0){
    infiniteRecursion2(x + 1);
  }
  else{
    System.out.println(x);
  }
}

private static void infiniteRecursion3(){
  infiniteRecursion3b();
}

private static void infiniteRecursion3b(){
  infiniteRecursion3();
}

private static void typeOverFlow1(){
  int x = Integer.MAX_VALUE;
  x++;
}

private static void typeOverFlow2(){
  int x = Integer.MAX_VALUE;
  int y = x + 1;
}

private static void InvalidCast1(){
  Object o = new Object();
  String s = (String)o;
}

private static void BadStringCmp1(){
  int x = 0;
  if("test1" == "test2"){
    x = 1;
  }
  else{
    x = 2;
  }
}

private static void BadStringCmp2(){
  int x = 0;
  String s1 = new String("test1");

  if(s1 == "test2"){
    x = 1;
  }
  else{
    x = 2;
  }
}

private static void BadStreamClose1(){
  try {
    FileInputStream x = new FileInputStream("z");
    x.close();
  }
  catch(Exception e){}
}
```

```
  private static void BadStreamClose2(){

    FileInputStream x;

    try{
      x = new FileInputStream("z");
    }catch(Exception e){}
    if(2 > 3){
      try{
        x.close();
      }catch(Exception e){}
    }
  }

  private static void arrayStoreException1(){
    Object x[] = new String[3];
     x[0] = new Integer(0);
  }

}

class DeadLock1{

  private static Object lock = new Object();
  private static Thread th1 = new Thread(){public void run(){f1(1);}};
  private static Thread th2 = new Thread(){public void run(){f2(2);}};

  public static void startThread(){
    th1.start();
    th2.start();
  }

  private static void f1(int num){
    synchronized (lock) {
      try {Thread.yield();} catch (Exception e) {}
      f2(num);
    }
  }

  private static synchronized void f2(int num){
    synchronized (lock) {f1(num);}
  }
}

class DeadLock2{
  private static Object lock1 = new Object();
  private static Object lock2 = new Object();
  private static Thread th1 = new Thread(){public void run(){f1();}};
  private static Thread th2 = new Thread(){public void run(){f2();}};

  public static void startThread(){
    th1.start();
    th2.start();
  }

  private static void f1(){
    synchronized (lock1) {
      try {Thread.yield();} catch (Exception e) {}
      synchronized (lock2) {f2();}
    }
  }
  private static synchronized void f2(){
    synchronized (lock2) {
      synchronized (lock1) {f1();}
    }
  }
}
```