Continuous Code-Quality Assurance with SAFE

Emmanuel Geay Eran Yahav Stephen Fink
IBM T.J. Watson Research Center
{egeay,eyahav,sjfink}@us.ibm.com
http://www.research.ibm.com/safe

ABSTRACT

This paper presents the design of SAFE (Scalable and Flexible Error Detection), a static analysis tool targeting lightweight program verification and bug finding for Java. The tool utilizes two types of analysis: a simple ”structural” checker based on pattern-matching, and an interprocedural flow-sensitive dataflow solver which integrates typestate checking and alias analysis. We describe how the tool integrates into a team development platform for analysis of batch builds, and user interface support built on the Eclipse platform.

1. INTRODUCTION

For many years, program analysis researchers have developed tools to partially verify program correctness and find potential bugs and coding problems. While most interesting program verification problems are formally undecidable, tools employ a variety of analysis techniques that can produce valuable results in practice.

A number of recent static analysis tools employ simple pattern-matching to find suspicious code patterns. Tools such as PMD [17] and FindBugs [13] have followed a mantra that “finding bugs is easy”; simple code scanning can identify many problems in code. Indeed, experimental results [20] show that simple techniques such as searching for likely mis-spellings in method names can yield useful information in real systems. Some tools (e.g.,[17, 5, 16]) provide simple pattern languages whereby a user can customize the tool’s checking for domain-specific problems.

While pattern matching can find some classes of bugs, researchers have also focused on deeper properties, such as security vulnerabilities [4] and API usage problems [18, 22]. These tools rely on powerful analysis techniques such as typestate verification [21], model checking [2], and sophisticated interprocedural dataflow and alias analysis [7, 11]. In many cases, scaling these techniques to large programs remains an open research topic. Nevertheless, these advanced tools have also yielded significant value in practical domains.

We describe SAFE (Scalable And Flexible Error detection), a new IBM Research project which incorporates both shallow and deep approaches for bug finding and lightweight program verification. This paper describes the design of the tool, and gives a high-level overview of the analysis techniques employed.

PEPM ’06 January 9–10, 2006, Charleston, South Carolina, USA.
Copyright 2006 ACM 1-59593-196-1/06/0001 ...$5.00.

SAFE is designed to run either in batch mode, as part of a continuous build/integration loop [6], or as an interactive component in an Eclipse development environment. This paper describes the design of the user interface and its integration in both usage scenarios.

The rest of this paper is organized as follows. In section 2, we describe the two main analysis components of SAFE — the structural and typestate checkers. In section 3 we present the various modes of SAFE execution. In section 4 we briefly describe some experiences with users. In section 5 we describe related work. Finally, in section 6, we discuss some future work.

2. ANALYSIS COMPONENTS

SAFE relies on two main analysis components:

- a *structural engine* based on (mostly) intraprocedural pattern matching. This engine tries to match suspicious known code patterns.
- a *typestate checking engine* performing (interprocedural) typestate checking with varying degrees of cost and precision, mostly depending on the way in which aliasing is handled.

Each engine allows the user to easily customize rules and properties based on application-specific constructs.

Structural Engine

The structural engine constructs an XML model of the program, and uses the model as a basis for evaluating queries defining common bug-patterns. Many existing bug-finding tools are based on pattern-matching over the program’s abstract syntax tree (AST) (e.g., PMD [17]). In contrast, SAFE constructs a model based on Java bytecode enriched by dataflow analyses such as constant propagation. This allows the user to define bug-patterns that encode semantic properties that cannot be expressed as AST patterns.

The SAFE user can write custom queries expressed as XPath, or more generally, XQuery. With these languages, users can write specifications that refer to application-specific types and names, without writing any Java code or modifying any SAFE code.

The structural engine currently handles patterns at two levels: (i) intraprocedural patterns specified over a representation of code and dataflow for a single procedure, and (ii) patterns specified over the structure of the program’s class hierarchy and individual classes.

Typestate Checking Engine

Typestate checking [21] is a well-known technique to check a variety of temporal properties; e.g., that an object is not used before it is initialized, or that a resource is not used after it has been closed.

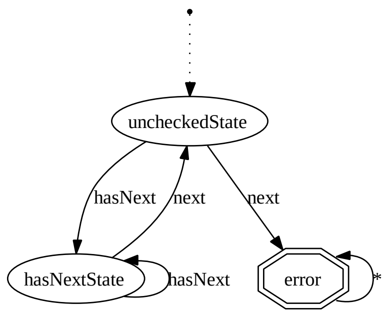


Figure 1: Specification of the property 'call `hasNext()` before calling `next()`' for the type `java.util.Iterator`.

Typestate checking models each object, at each point during a program's execution, as occupying one of finitely many states. The operations permitted on an object depend on its state, and each operation may alter the state of the object. Typestate checking tries to statically determine if the execution of a given program may cause an operation to be performed on an object in a state where said operation is forbidden.

In SAFE, the user specifies a typestate property by providing (via XMI) a deterministic finite state automaton (DFA) defining the permitted sequences of operations for a type. For example, the DFA of Fig. 1 defines some of the permitted sequences of operations for the type `java.util.Iterator`. In particular, this DFA specifies the requirement that for an `Iterator` object, a call to the `next()` method must be preceded by a call to `hasNext()`. If any `Iterator` object in the program violates this requirement, the SAFE typestate engine will report a finding.

A key challenge for precise typestate checking concerns the treatment of aliasing. Various flow-insensitive alias analyses have been shown to scale to large programs [12, 15]. However, in many cases, typestate checking requires the ability to perform "strong updates" [3], which requires flow-sensitive treatment of pointers.

We have developed a family of integrated typestate and alias analysis algorithms of varying cost and precision. These algorithms track aliases with flow-sensitive and context-sensitive symbolic access paths, but only at program points where dataflow information indicates that pointer manipulations will affect the typestate property. In this way, the algorithms focus the expensive alias analysis to small portions of the program.

The SAFE typestate checker provides a *staged* analysis, running a sequence of integrated alias and typestate checkers applied in order of increasing precision and cost. Each stage checks only statements that previous stages failed to verify, thus reducing work for the most expensive algorithms.

A detailed description and evaluation of the typestate algorithms falls beyond the scope of this paper, but will appear in an upcoming report [9].

3. SAFE MODES

SAFE has been designed to run in a variety of contexts: in a continuous integration [6] loop, as an Eclipse IDE plugin, as an ANT [1] task, and as a command-line program. With these various modes, a SAFE user can run the relatively inexpensive analyses interactively in the IDE, and the build system can run the deeper expensive analyses in batch.

3.1 SAFE within CruiseControl

With the current state-of-the-art, aggressive interprocedural analyses can run for hours, and so best suit a background or batch environment. Also, in a team development environment, these interprocedural analyses may produce the most value checking builds for integration problems, rather than for intra-module problems in a single developer's workbench.

For these reasons, we have integrated SAFE into CruiseControl [6] (as shown in Fig. 2), an environment for *continuous integration* [10]. SAFE runs as part of the normal CruiseControl build loop, and publishes results to an HTML tab in the CruiseControl build results page. A team using CruiseControl can install SAFE on the build server with no overhead for individual developers.

3.2 SAFE within Eclipse

While a batch SAFE runner provides a low-overhead means to introduce SAFE into a project, static analysis can also provide value for an individual developer's IDE. With analysis in the IDE, the developer can analyze code for problems *before* committing to the source code repository. Additionally, the IDE provides for a richer, interactive user interface. For these reasons, SAFE supports deployment as a component for the popular Eclipse Java Development Environment.

Detecting Entry Points for Whole-Program Analysis

SAFE's "whole-program" interprocedural analyses require a specification of the program's entry points, that is, possible starting points for program execution. While in some cases a program has a single entry point (its `main` method), many cases require specification of multiple entry points (e.g., libraries, Eclipse plugins, incomplete code).

SAFE allows the user to manually specify entry-points through an XML file. SAFE can alternatively detect entry points automatically. For example, for standard Java projects (projects with a "Java nature"), SAFE automatically selects all `main` methods as candidate entry points. Similarly, SAFE provides support to analyze J2EE programs according to the remote interfaces exported by the Enterprise Java Beans.

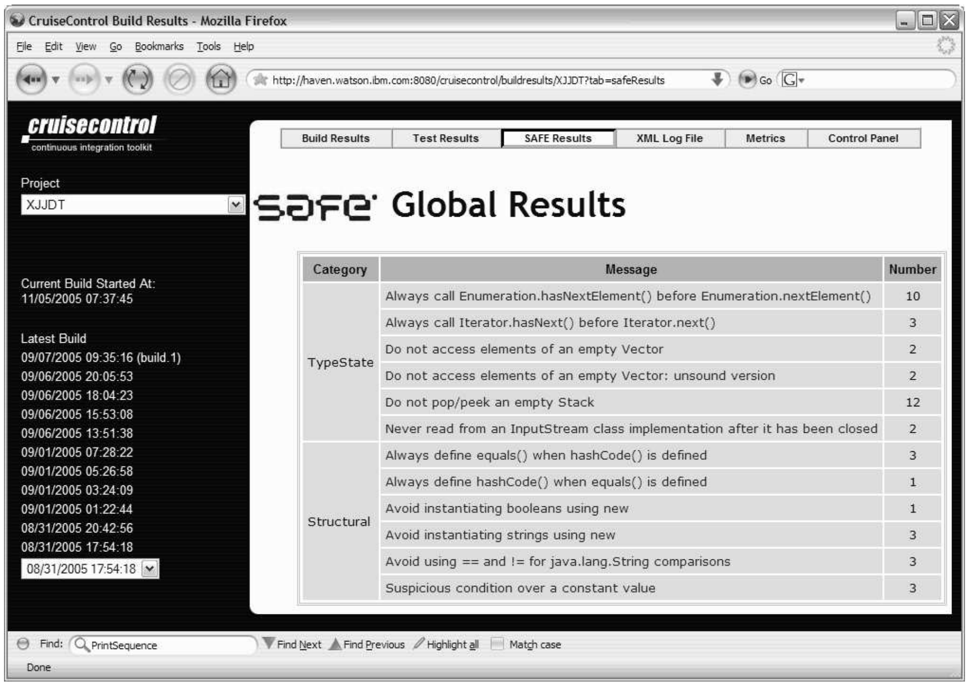
Unlike stand-alone Java programs, Eclipse plugins (projects with a "plugin nature") do not contain a `main` method. The two common techniques for running code in a plugin are through its *plugin main class* definition (with its `start` and `stop` methods), and by using a callback mechanism (*extension points*). SAFE deduces potential plugin project entrypoints by collecting all classes that can be instantiated by reflection in the extension points of the selected projects and their dependencies.

Visualization of SAFE Findings

In Eclipse, SAFE provides an *Analysis Results* view which displays all analysis findings categorized by rules. Double-clicking on a finding has two effects: opening a Java editor at the source location corresponding to the finding, and opening a *Rule Details* view. The rule details view provides additional information about the finding such as its textual description, an example code fragment, and a description of suggested corrective actions.

Fig. 3 shows a screenshot of SAFE within Eclipse. In this figure, the *Analysis Results* view appears at the bottom, where a finding for the typestate rule "Always call `Iterator.hasNext()` before calling `Iterator.next()`" has been selected. The upper part of the figure shows the corresponding source line highlighted in the Java editor. The *Rule Details* view provides additional information about the finding at the right. Finally, the pop-up dialog box blocking part of the view shows how analysis configurations are defined.

SAFE uses the Eclipse platform to provide content for online



CruiseControl Build Results - Mozilla Firefox

http://haven.watson.ibm.com:8080/cruisecontrol/buildresults/XJJDT?tab=safeResults

Project: XJJDT

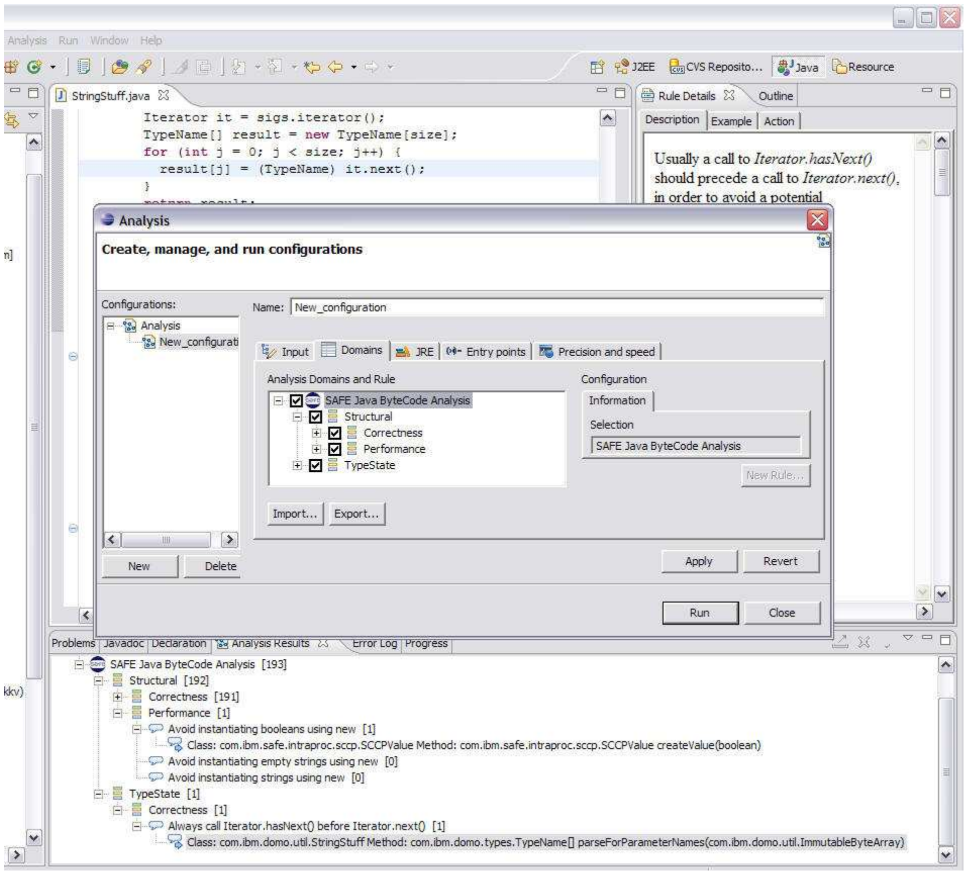
Current Build Started At:
11/05/2005 07:37:45

Latest Build
09/07/2005 09:35:16 (build.1)
09/06/2005 20:05:53
09/06/2005 18:04:23
09/06/2005 15:53:08
09/06/2005 13:51:38
09/01/2005 07:28:22
09/01/2005 05:26:58
09/01/2005 03:24:09
09/01/2005 01:22:44
08/31/2005 20:42:56
08/31/2005 17:54:18

safe Global Results

Category	Message	Number
TypeState	Always call Enumeration.hasNextElement() before Enumeration.nextElement()	10
	Always call Iterator.hasNext() before Iterator.next()	3
	Do not access elements of an empty Vector	2
	Do not access elements of an empty Vector: unsound version	2
	Do not pop/peek an empty Stack	12
	Never read from an InputStream class implementation after it has been closed	2
Structural	Always define equals() when hashCode() is defined	3
	Always define hashCode() when equals() is defined	1
	Avoid instantiating booleans using new	1
	Avoid instantiating strings using new	3
	Avoid using == and != for java.lang.String comparisons	3
	Suspicious condition over a constant value	3

Figure 2: SAFE within CruiseControl.



```
Iterator it = sigs.iterator();
TypeName[] result = new TypeName[size];
for (int j = 0; j < size; j++) {
  result[j] = (TypeName) it.next();
}
```

Usually a call to *Iterator.hasNext()* should precede a call to *Iterator.next()*, in order to avoid a potential

Analysis

Create, manage, and run configurations

Name: New_configuration

Analysis Domains and Rule
- [✓] SAFE Java ByteCode Analysis
  - [✓] Structural
    - [✓] Correctness
    - [✓] Performance
  - [✓] TypeState

Configuration
Information
Selection: SAFE Java ByteCode Analysis

```
SAFE Java ByteCode Analysis [193]
  Structural [192]
    Correctness [191]
    Performance [1]
      Avoid instantiating booleans using new [1]
        Class: com.ibm.safe.intraproc.sccp.SCCPValue Method: com.ibm.safe.intraproc.sccp.SCCPValue createValue(boolean)
      Avoid instantiating empty strings using new [0]
      Avoid instantiating strings using new [0]
  TypeState [1]
    Correctness [1]
      Always call Iterator.hasNext() before Iterator.next() [1]
        Class: com.ibm.domo.util.StringStuff Method: com.ibm.domo.types.TypeName[] parseForParameterNames(com.ibm.domo.util.ImmutableByteArray)
```

Figure 3: SAFE within Eclipse.

Benchmark	Classes	Methods	Bytecode lines	Time (mm:ss)
PDE	1,128	6,627	3,2261	1:37
RCP	4,144	32,883	1,567,979	5:03
JDT	4,859	41,325	2,446,541	8:11
Platform	14,386	112,195	5,117,868	20:55

Table 1: Structural engine benchmarks for Eclipse projects and analysis running times.

Rule	PDE	RCP	JDT	Platform
Always define hashCode() when equals() is defined	6	21	37	80
Always define equals() when hashCode() is defined	0	2	17	5
Avoid calling finalize() explicitly	0	0	0	1
Avoid calling System.gc()	0	3	3	8
Avoid calling Thread.stop()	0	0	0	1
Avoid instantiating booleans using new	2	16	5	114
Avoid instantiating empty strings using new	1	8	11	25
Avoid instantiating strings using new	0	2	8	26
Avoid using == and != for String comparisons	6	37	26	154
Avoid using Object.notify()	0	5	1	35
Potential infinite recursion	0	1	4	11
Potential null dereference	0	0	0	1
Suspicious condition over a constant value	3	54	39	186
Suspicious equal() method has been defined	1	0	1	0
Total	**18**	**149**	**152**	**647**

Table 2: Sample structural rules and findings for Eclipse benchmarks.

help, and an Eclipse update site for easy web-based installation.

3.3 Other modes

In addition to CruiseControl and Eclipse, SAFE also supports execution as an Ant task or as a stand-alone command-line tool. In these modes, the user provides the analysis scope and options as configuration XML files and/or command-line options, and SAFE produces the analysis results in an XML file.

For these modes, SAFE supplies a web-based UI for exploring the results via a browser, using Extended Style Sheets and Javascript scripts. Several IBM groups instead read SAFE XML output directly, for use with other tools.

Finally, in order to use Eclipse-based functionality (such as automatic detection of entry points) in a batch process, we also support running SAFE in an Eclipse *headless mode*. Technically, in this mode SAFE becomes an Eclipse Rich Client Platform (RCP [19]) application, without a UI, but with the underlying eclipse data-model. To run SAFE in this mode, the user provides a launch configuration and the location of an Eclipse workspace.

4. EXPERIENCE WITH SAFE

SAFE is being used by a number of early adopter organizations across the IBM Corporation. One team has integrated SAFE into its build process; other teams mostly run it manually when required, either at the command-line or from within Eclipse. We have been running SAFE via CruiseControl on a number of IBM product code bases, and reporting bugs back to development teams. Some of the reported bugs have been confirmed as defects. Additionally, we use SAFE on our own build servers in IBM Research, continuously checking several research projects.

Name	Description
Enumeration	Call `hasNextElement` before `nextElement`
InputStream	Do not read from a *closed* `InputStream`
Iterator	Do not call `next` without checking `hasNext`
KeyStore	Always initialize a `KeyStore` before using it
PrintStream	Do not use a *closed* `PrintStream`
PrintWriter	Do not use a *closed* `PrintWriter`
Signature	Follow initialization phases for `Signature`s
Socket	Do not use a `Socket` until it is *connected*
Stack	Do not `peek` or `pop` and empty `Stack`
URLConn	Illegal operation when already connected
Vector	Do not access elements of an empty `Vector`

Table 3: Sample typestate properties.

In addition to real projects using SAFE, we have also evaluated the structural engine over 4 Eclipse components: Platform, PDE, RCP and JDT. The sizes of these projects in terms of classes, methods, and number of bytecode statements, are shown in Table 1. The findings reported by SAFE for these benchmarks are shown in Table 2.

All experiments ran on IBM Intellistation Z pro with two 3.06 GHz Intel Xeon CPUs, and 3.62 GB of RAM running Windows XP with IBM J2RE 1.4.2.

We manually verified some of the findings reported by SAFE, and in particular, found a real null pointer exception in the update-manager code of the Platform component.

The rule "suspicious condition over constant value" returns a large number of matches. This is mostly due to various debugging flags that guard code producing output to a tracing facility.

The matches returned by the rule "potential infinite recursion" are all false alarms. This is due to code that follows a proxy design pattern and delegates responsibilities to other objects that implement the same interface. Unfortunately, the current version of the structural engine is unable to observe that the target object of the call is different than the calling object.

Table 3 shows some of the typestate properties checked by SAFE. Another report [9] presents a detailed evaluation of the typestate checkers; the results show that the typestate checkers verified the absence of typestate violations for over 95% of the eligible statements, over a suite of 19 moderate-sized projects (up to 100,000 lines of code) We are currently working on algorithmic improvements to scale the typetate checker to larger code bases, and also to analyze libraries in addition to complete programs.

5. RELATED WORK

Due to space limitations, we review here only a few of the most relevant related projects.

FindBugs [13] is Java bytecode pattern-matching tool with patterns corresponding to error-prone constructs. The functionality and limitations of FindBugs resemble those of our structural checker. However, FindBugs operates directly on Java bytecode, where SAFE operates on an intermediate representation enriched by information from dataflow analysis. In addition, the SAFE structural engine allows flexible pattern specification via XPath and XQuery.

A large number of tools operate directly on the program's AST, including CodeReview [5], JTest [16], JiveLint [14], and PMD [17]. Some of these tools checking compliance with various coding styles. CodeReview and JTest provide refactoring operations to correct problems, integrated with Eclipse.

ESP [7] is a typestate checker for C programs, utilizing selective path-sensitivity for scalability. ESP handles aliasing by taking a two-phased approach in which typestate checking is preceded by a pointer analysis. In contrast, SAFE supports multiple ways of handling aliasing, including an integrated typestate analysis and alias analysis. In the future, we plan to extend SAFE to use selective path-sensitivity in a way similar to ESP.

6. FUTURE WORK

In the near future, we are focusing our efforts on improving the usability and scalability of SAFE. The structural engine currently analyzes code bases of several million lines, while we can currently scale a typical typestate analysis up to a few hundred thousand lines of code. To improve usability, we will add various features to allow user to customize the reported findings. We also plan to incorporate a symbolic analysis for generating counter-examples from SAFE findings, and to reduce the rate of reported false positives. In addition, we are investigating the use of specification mining techniques to automatically generate typestate specifications.

7. REFERENCES

[1] Apache Ant. http://ant.apache.org/.

[2] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[3] D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[4] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA, 2002. ACM Press.

[5] CodeReview. http://www-128.ibm.com/developerworks/rational/library/05/higgins.

[6] CruiseControl. http://cruisecontrol.sourceforge.net.

[7] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 57–68, June 2002.

[8] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Proc. of SAS'03*, volume 2694 of *LNCS*, pages 439–462. Springer, June 2003.

[9] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Typestate checking in the presence of aliasing. in preparation, 2005.

[10] M. Fowler. Continuous Integration. http://www.martinfowler.com/articles/continousIntegration.html.

[11] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, New York, NY, USA, 2002. ACM Press.

[12] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. 36(5):254–263, May 2001. In *Conference on Programming Language Design and Implementation (PLDI)*.

[13] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM Press.

[14] JLint. http://www.sureshotsoftware.com/javalint.

[15] O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. In *12th International Conference on Compiler Construction (CC)*, volume 2622 of *LNCS*, pages 153–169, Apr. 2003.

[16] Parasoft JTest. http://www.parasoft.com/jsp/products/home.jsp?product=Jtest.

[17] PMD. http://pmd.sourceforge.net/.

[18] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. Conf. on Prog. Lang. Design and Impl.*, volume 37, 5, pages 83–94, June 2002.

[19] RCP. http://www.eclipse.org/rcp.

[20] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *ISSRE '04: Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering*, November 2004.

[21] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[22] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, New York, NY, USA, 2004. ACM Press.

Acknowledgements

Many thanks to Steve Gutz for his contributions to the SAFE Eclipse environment.