

A Framework for the Static Verification of API Calls^{1,2}

Diomidis Spinellis² and Panagiotis Louridas

*Department of Management Science and Technology
Athens University of Economists and Business
Patission 76
GR-104 34 Athens, Greece
email: {dds, louridas}@aueb.gr*

Abstract

A number of tools can statically check program code to identify commonly encountered bug patterns. At the same time, programs are increasingly relying on external APIs for performing the bulk of their work: the bug-prone program logic is being fleshed-out, and many errors involve tricky subroutine calls to the constantly growing set of external libraries. Extending the static analysis tools to cover the available APIs is an approach that replicates scarce human effort across different tools and does not scale. Instead, we propose moving the static API call verification code into the API implementation, and distributing the verification code together with the library proper. We have designed a framework for providing static verification code together with Java classes, and have extended the FindBugs static analysis tool to check the corresponding method invocations. To validate our approach we wrote verification tests for 100 different methods, and ran FindBugs on 6.9 million method invocations on what amounts to about 13 million lines of production-quality code. In the set of 55 thousand method invocations that could potentially be statically verified our approach identified 800 probable errors.

Key words: Static analysis, API, Library, Programming by contract, FindBugs

¹ *Journal of Systems and Software*, 80(7):1156–1168, July 2007.

² This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

³ Corresponding author.

1 Introduction

Automatic program verification tools have had a significant impact on software development, and are more and more used in practice to eliminate many errors that in the past would have caused program crashes, security vulnerabilities, and program instability (Johnson, 1977; Bush et al., 2000; Ball and Rajamani, 2002; Das et al., 2002; Csallner and Smaragdakis, 2005; Cok and Kiniry, 2005; Barringer et al., 2006). However, two software development trends are now hindering the applicability of automated program verification tools:

- (1) the increasing use of binary-packaged components (for the most part libraries) through their application programming interface (API), and
- (2) the increasing API sophistication, and in particular the embedding of many different domain-specific languages (DSLs) as strings in the program code.

Both trends reduce the efficiency of the current approaches. The use of feature-rich libraries in their binary form handicaps verification programs that require access to source code, such as ESC/Java (Flanagan et al., 2002), and also programs that contain a fixed-set of specific bug patterns, like ITS4 (Viega et al., 2000). Furthermore, the diversity of the libraries handicaps any tool that depends on a centralized repository of verification patterns. In addition, the embedding of DSLs, like SQL and XPath, in strings appearing in the program's source code can introduce bugs that are beyond the reach of the current breed of tools based on approaches like theorem proving (Flanagan et al., 2002), dataflow analysis (Jackson, 1995), and finite state machines (Ball and Rajamani, 2002). To overcome these difficulties we propose a framework for incorporating API call verification code within each library containing the corresponding API implementation. Through the use of reflection techniques program checkers can invoke this code and verify that the actual arguments presented to API invocations meet corresponding value constraints.

1.1 Programming through APIs

An application programming interface specifies how one software component or system can communicate with a provider of some services. The communication can take the form of a (local or remote) procedure call, a method invocation, an operating system trap, or a web service invocation. The provider of the API functionality can be packaged in the form of a library, a component (Szyperski, 2002), a running process, or an abstract service available over the internet. Widely-used APIs include those defined in the Single UNIX Specification, the Microsoft Windows Win32, ODBC, and .NET APIs, the APIs defined in the Java 2 Enterprise Edition platform, as well as vertical APIs addressing domains such as graphics rendering (OpenGL, DirectX), storage devices (ATAPI, SCSI), and network interfaces (NDIS).

Project name	Classes			Foreign
	Total	Own	Foreign	%
Centraview 2.0.6	5,813	2,219	3,594	61.8
Compiere 2.5.2d	11,162	1,611	9,551	85.6
Eclipse 3.1	18,584	16,227	2,357	12.7
Jahia 4.1.0	17,310	1,308	16,002	92.4
JBoss 4.0.2	37,308	10,235	27,073	72.6
Netbeans 4.1	14,876	7,963	6,913	46.5
OpenWFE 1.5.4	6,268	444	5,824	92.9

Table 1

The use of foreign classes in Java projects

To substantiate the need for a program verification approach specifically targeting API calls we must look into the *number* and *size* of existing APIs, their actual *utilization* in real-life software, the organizational structure of library development, and trends regarding their use. We gathered data from three sources:

- the FreeBSD ports collection, a set of more than 10,000 contributed applications, mostly written in C/C++, and organized in a way that allows straightforward installation,
- a number of large and popular Java projects,
- historical implementations of the Emacs editor.

By and large, our results indicate that the number of projects that use external APIs—that is, APIs that are not by default part of their language or execution environment—are significant, as is the number of these APIs, and their size. By tracking the dependencies of the FreeBSD packages we established that the 12,357 ports packages in our FreeBSD 4.11 system, had in total 21,135 library dependencies; i.e., they required a library, other than the 52 libraries that are part of the base system, in order to compile. The library dependencies comprised 688 different libraries, while the number of different external libraries used by a single project varied between 1 and 38, with a mode value of 2. Furthermore, 5,117 projects used at least one external library and 405 projects used 10 or more.

We tracked the use of foreign APIs in some large representative Java projects by analyzing all the Java archive files comprising the project’s binary distribution, and categorizing the class files they included according to their package. Those whose package was the same as the project (for example, `org.eclipse` for Eclipse) were categorized as “own”, the rest as “foreign”. The results are summarized in Table 1. Note that the numbers in the table do not include the Java runtime classes, because these are by default part of the runtime environment. Evidently, many projects depend on library code for a large part of their functionality.

Product	Release	Size	Imported
	year	(kLOC)	elements
GNU Emacs 18.59	1992	94	121
XEmacs 19.16	1997	232	406
jEdit 4.3	2005	145	2927

Table 2

The use of API elements over time

The numbers we collected also point toward a highly decentralized organizational structure under which libraries are developed. The 111,321 foreign classes appearing in Table 1 are not unique, because the same classes may be used in multiple projects or subprojects; in our data set the class `org.apache.commons.logging.LogSource` was used 20 times. Nevertheless, the projects use among them 60,273 unique classes outside their own domain. Most of the classes belong to packages named according to Sun’s conventions: the name’s first elements define the organization behind the package. By looking at the first two elements of the package names we found 66 different entities behind the packages, like `com.ibm` or `org.jboss`. Clearly, any proposal for handling API call verification must take this diversity into account.

The previous two paragraphs indirectly support our claim regarding the size of existing APIs through the large number of foreign classes used by the Java projects. We can further substantiate the size of existing APIs by looking at the number of functions and methods available in some modern environments.

- The Single UNIX Specification version 2 identifies as interfaces 725 functions and macros.
- The Windows API list distributed with Visual C/C++ 5.0 contains 3,777 elements listed as DLL functions.
- The Microsoft .NET Framework 1.1 documents 3,136 types and 15,724 methods.
- The Java 1.5.0 runtime environment has 6,520 public classes that contain 52,743 public methods and constructors.

In fact, such is the size and complexity of modern APIs, that a recent paper proposes a system for mining API usage patterns from a corpus of sample programs in order to aid programmers in the navigation of the increasingly large and convoluted APIs (Mandelin et al., 2005).

Finally, anyone who has been writing software for the last 20 years will readily attest that software systems are being fleshed-out, increasingly using third-party components through API calls. It is instructive to witness this trend in action. Table 2 details the total number of imported functions or methods used by three different editors. The text-based GNU Emacs editor derives from a 1985 codebase, and provides a feature-rich editing environment while using just 121 elements from the

operating system and the C library. Released about 10 years later, XEmacs uses 406 elements to provide an X Window System GUI, while nowadays jEdit uses 2927 Java methods to provide a similar interface with almost 100 thousand lines less code.

1.2 Domain-specific Languages

A domain-specific language is tailored specifically to an application domain: rather than being general purpose its aim is to capture precisely a domain's semantics. Examples of domain-specific languages are BNF grammar specifications and regular expressions, as used for example by the *yacc* and *lex* tools for generating lexical analyzers and parsers (Johnson and Lesk, 1987), the SQL database definition and manipulation language, and XSLT, the XML document transformation language. Often DSL fragments are directly embedded into the source code of a general purpose language (Spinellis, 2001), in many cases as strings. Apart from the ubiquitous SQL statements found in any typical database client, other DSLs that appear as strings in code are regular expressions, output formatting specifications, various applications of XML, XPath queries, and URLs. The verification of code written in these DSLs is both important and worthy of an explicitly targeted approach.

Although some of the DSLs we have described may appear trivial, they are not. Even a URL is defined by a fairly extensive syntax, and specific URL schemes can have very precise rules for the elements that appear in them. Java's Generic Connection Framework defines a BNF grammar for a number of schemes, allowing for example the connection to a Bluetooth GPS receiver using a URL like

```
btsp://000A5600F776:1;authenticate=false;encrypt=false;
master=true;authorize=true
```

The name value pairs in the above URL are precisely defined and a spelling error in one of them would result in a runtime error. In fact, a spelling mistake is not the only error that can occur within a DSL string embedded in a general purpose language. Other error classes include the following.

Syntax Error Some DSLs, like SQL, are defined by an extensive syntax, and it is easy for a programmer to write an invalid statement.

Internationalization Problem Regular expressions and format strings, both defined through a DSL, can often contain assumptions that can make a program difficult to localize. As an example, a regular expression containing the sequence `[A-Z]` to specify an uppercase letter will only work with ASCII characters, and should probably be changed to specify a Unicode category through a sequence like `\p{Lu}`.

Portability Problem Like general purpose programming languages, many of the DSLs have been implemented or extended in a number of non-standard and in-

compatible ways. A programmer may unwittingly use such an extension, and thus burden the program with unintended portability restrictions that will cause problems in the future. As an example, our prototype tool flagged the following SQL statement appearing in OpenWFE as wrong.

```
SELECT workitem_id, action, arg
FROM action where msg_err is null
```

Although not readily apparent, the error in the above statement is the use of `action`, an SQL reserved word, as an identifier.

All the DSLs we have outlined are beyond the reach of general purpose program verification tools, because they appear in the program code as untyped strings. Errors in the DSLs can, and are, caught by special purpose approaches that target the specific DSLs. However, such schemes are inherently difficult to scale: they must incorporate special-purpose checking code for every DSL in existence. Furthermore, the special-purpose verification code may well end-up duplicating functionality available in the actual DSL implementation, such as a parser. For these reasons we believe that API verifiers should not be implemented as part of verification tools, but should be incorporated into the API libraries.

2 Research Context

Our approach to program verification falls within the domain of static program analysis. This involves the analysis of program code for certain properties without executing it; usually, it is performed at compile time. Errors discovered late cost much more than errors discovered early in the development process (Fagan, 1976). Static analysis aims at lowering development costs by eliminating problem spots as early as possible.

Before we examine static program analysis methods, let us note that a different but complementary approach for verifying code across component boundaries involves **run time monitoring** to check for conditions that are more stringent than those specified in a procedure’s API type signature. These are typically based on the application of lightweight formal methods during the execution of programs. See for example the work of Marinov and Khurshid (2001); Havelund and Roşu (2004); Artho et al. (2005); Pnueli et al. (2006) and the annual *Workshop on Runtime Verification* (Barringer et al., 2006). Another runtime monitoring technique involves “debugging” library implementations that perform more stringent tests on their arguments at the expense of reduced time and space performance. This approach is often used in languages that allow unbounded pointers, like C and C++. Examples include `malloc` debug libraries (Barach et al., 1982), the GNU implementation of the C++ STL functionality, and the Microsoft Windows SDK debug libraries. As we argue in our concluding remarks, runtime monitoring tools complement our approach with a potential to locate more errors slightly later in the development

cycle.

The progenitor of static bug finding tools is Lint (Johnson, 1977). Lint employs heuristics for locating bugs, portability problems and style deviations in C source code. Lint's success led to many projects with similar goals that follow a variety of different approaches. A comparison of bug finding tools for Java can be found in the work by Rutar et al. (2004).

Heuristics-based approaches are used by a number of successors to Lint, like LCLint (Evans et al., 1994), which checks C programs against a set of formal specifications written in the LCL language (Guttag and Horning, 1993), and Splint (Evans and Larochelle, 2002), LCLint's successor. JLint (JLint, 2004) finds a fixed set of synchronization, inheritance, and dataflow problems in Java programs. Heuristics are also employed by ITS4 (It's The Software Stupid! Security Scanner) (Viega et al., 2000), which statically scans security-critical source code for vulnerabilities. ITS4 acts as a smarter version of the Unix `grep` command by locating, for example, calls to unsafe library functions like `strcpy` and `gets`. Metacompilation (MC/Coverity, Hallem et al. (2002)) finds bugs in C programs by using heuristics written as compiler extensions (checkers) that are executed by an analysis engine. Checkers can be written to find violations of known correctness rules and even automatically infer such rules from source code (Engler et al., 2001). SABER (Reimer et al., 2004) performs static analysis of Java programs looking for violation of programming rules, grouped in categories; its matches are subsequently filtered to reduce false positives.

In this area Microsoft's PREfast (Larus et al., 2004) tool parses functions and invokes plugins that traverse the function's parse tree, looking for idioms that might indicate problem points. Modern versions of the Microsoft Windows API header files annotate the declaration of many function parameters to specify whether a parameter passes values in or out of the function, whether it represents a buffer, or whether it specifies a buffer's size. The declaration annotations are based on primitives of the so-called *Standard Annotation Language* (SAL).

Popular heuristics-based tools for Java are PMD (Copeland, 2005) and FindBugs (Hovemeyer and Pugh, 2004). Some of the tools based on heuristics can detect errors in DSL strings. For example, the GNU C compiler can verify `printf` format specifications, while FindBugs will attempt to parse regular expressions.

A different class of checkers uses **theorem proving** techniques. ESC/Java (Extended Static Checking for Java) (Flanagan et al., 2002) bases its checking on tag-like annotations added on program comments. Annotations, state routine preconditions, postconditions, object invariants, and ghost fields (fields only seen by the checker, and not by the compiler), are written in JML (Java Modeling Language) (Leavens et al., 2005; Burdy et al., 2005). Experience showed that about 40–100 (manually inserted) annotations are required per thousand lines of code. The bur-

den can be lightened if annotations are produced automatically; on this path, Houdini (Flanagan and Leino, 2001), is an annotation assistant. Another extension to ESC/Java is Check 'n' Crash (CnC) (Csallner and Smaragdakis, 2005). CnC takes ESC/Java's output and constructs a series of JUnit tests (Beck and Gamma, 1998). Recently, the ESC/Java and JML research efforts appear to have been merged under the roof of ESC/Java2 (Cok and Kiniry, 2005).

The preconditions, postconditions, and object invariants that theorem proving tools use are not a new concept. They characterize Programming by Contract, as formalized by VDM (Jones, 1990) and exemplified in the Eiffel programming language (Meyer, 1997). The difference is that Programming by Contract is typically associated with the testing of invariants at runtime, whereas static checking aims at checking them at compile time.

Conceptually related to theorem proving is an approach taken by the functional programming community that builds on **soft type checking** (Wright and Cartwright, 1997). Functional programming languages are sometimes dynamically typed; soft type checking uses an inference engine (without annotations) to deduce types for the variables used in a program. The approach has been used in a Scheme implementation (Findler et al., 2002) and in Erlang for checking commercial telecommunications software (Lindahl and Sagonas, 2004).

Aspect (Jackson, 1995) builds on **dataflow analysis**. It employs assertions that annotate program code to indicate dependencies between variables used in procedures. Analysis then indicates places where variable use does not follow the specifications; for instance, errors of omission, when the value of a variable does not depend on a variable declared as a dependency. Aspect has been used for checking CLU programs with a specification to code ratio varying from 25% to 50%.

Finite state machines have had a number of followers in the field. Inspired by the success of finite state machines in hardware design, Bandera (Corbett et al., 2000) uses program slicing (Weiser, 1981) and abstraction to generate a tractable finite state machine that checks specifications. Symbolic model checking using finite state machines is used by SLAM (Ball and Rajamani, 2002), ESP (Error detection via Scalable Program analysis) (Das et al., 2002), and MOPS (MOdel Checking Programs for Security properties) (Chen and Wagner, 2002), which models the program as a pushdown automaton and security properties as finite state automata. In particular, Ball and Rajamani (2002) used the SLAM toolkit specifically for checking the (temporal) properties of interfaces, while Chaki et al. (2003) targeted the MAGIC tool to software components written in C. SLAM was adopted by Microsoft as a static driver verifier (Ball et al., 2004). Fugue (DeLine and Fähndrich, 2004), another Microsoft tool, employs both dataflow analysis and state machines. Fugue is a static protocol checker for languages that compile to Microsoft's CLR (Common Language Runtime). The user annotates the program source to indicate state transitions and custom checking code to be executed before and after method exe-

cution.

Abstract interpretation (Cousot and Cousot, 1977) can also be a basis for static analysis. Abstract interpretation builds and reasons upon an abstract model of a program; the absence of errors in the model implies the absence of some types of errors in the concrete program. Abstract interpretation has been used in the context of safety-critical software (Blanchet et al., 2003).

PREFIX (Bush et al., 2000) adopts a related technique, **symbolic execution**. It parses the source code into abstract syntax trees, and then follows a program's behavior by emulating function calls using automatically generated function models. A function model embodies the behavior of a function in a Lisp-like notation.

A different approach for avoiding program errors involves creating a **new languages** that enforce safety properties. Cyclone is a safe dialect of C (Jim et al., 2002). The Cyclone compiler refuses to compile any valid C program whose safety it cannot guarantee through static analysis and insertion of runtime checks. Vault (DeLine and Fähndrich, 2001) is a C-like language with annotations that express a function's preconditions and postconditions, and also track program resources such as files.

All the approaches we have outlined have the potential to contribute to a program's verification. However, none of the static checking approaches we described explicitly targets its checking across the division between application code and third party libraries. In particular, because third-party libraries are typically used in their compiled form, it is difficult to apply on them tools that rely on the availability of source code. First of all, notwithstanding the move toward open source software, there will probably always be companies which will distribute their software in binary form to protect their proprietary information. For example, although Linux is an open source project many hardware vendors are currently distributing their Linux drivers in binary form. It has even been suggested that some vendors are adopting a binary-only distribution policy to minimize the risk of patent-related lawsuits. In addition, even open source projects often use third party libraries in binary form for the sake of convenience; building a library from source may require extra tools, can be a lot of work, and can introduce an additional source of instability.

Other advantages of the framework we will describe over comparable approaches are the provision to API developers of a facility to allow the static verification of function arguments, its ability to verify DSL code embedded in function arguments, the potential to reuse existing API implementation code for verification purposes, the wide range of verification checks that can be supported, and the fact that it is an open framework on which others may freely build upon.

3 Adjunct Verification Code

Having established in Section 1.1 that programs increasingly utilize complicated APIs from a large and diverse set of third-party libraries, it is easy to see that API-specific verification code should be tied to the library providing the actual implementation. We therefore propose that every API implementation should carry with it functionality for the static verification of API calls at compile time. Verification tools can then tap into this functionality to extend their reach into the—now often opaque to them—API invocations.

There are two approaches for tying verification code with an API. One can use a *declarative* formalism, like that adopted by JML and ESC/Java2 (Cok and Kiniry, 2005). For example a queue data structure *enqueue* operation can be annotated with the following statements.

```
@ requires \typeof(e) <: elementType
@ modifies size;
@ ensures size == \old(size) + 1;
```

Although in the verification tools we mentioned the annotations are embedded in comments, and therefore inaccessible to programs working with binary-distributed libraries, the annotations could conceivably be placed in the compiled object’s data section and fetched from the library at compile time. The other approach involves adding in the library *imperative* code that will verify a given set of arguments encountered at compile time for correctness.

Both approaches have benefits and drawbacks. The declarative approach can be used together with automated theorem proving techniques and can therefore catch errors that are beyond the reach of the imperative checks. The imperative approach can be readily understood by developers not versed in formal specifications—unfortunately the norm in the software industry—and therefore easier to implement in a decentralized fashion: separately by each API developer. Moreover, the DSL verification problems we described earlier are a lot easier to handle in the imperative approach. Therefore, taking a mainly strategic decision, we decided to start by designing and implementing a framework around imperative functions that can verify constant API arguments. If routine API verification catches on, developers can be enticed to enhance their verification specifications with more extensive and precise declarative annotations.

The API verification framework we propose adds to every API method, constructor, or function, an *adjunct* verification method or function. This element has a name and argument signature close to that of the API element it is used to verify, but is suitably differentiated to avoid name clashes and code bloat. As an example, in our prototype Java implementation the verifier for the function `java.lang.Math.log` is `java.lang.MathV.log`. The verification function receives as

its arguments the arguments to the actual function (a method's invocation target is passed as the first argument), along with a boolean array that indicates which values could be determined at compile-time, and which could not. It returns as a result one of the following values.

Unverified None of the method's arguments could be verified; all the arguments have values that can not be determined at compile time, and at least one of the arguments could cause the method to fail at runtime.

Partially Verified A method's target object or some of the method's arguments are correct.

Arguments Verified All the method's arguments are correct.

Correct The method's invocation target and its arguments are correct. The invocation can not result in a runtime error related to the invocation target and its arguments.

Argument Error A discrete argument's value is incorrect. Example: a negative string starting position is passed to a substring function.

Domain Error A continuous value is outside its allowable domain. Example: a value greater than 1 is passed to an arc cosine function.

Name Error A specified name is incorrect. Example: a non-existent character set name is passed to a character code translation function.

Syntax Error A (DSL) argument's syntax is wrong. Example: the table name is missing from an SQL SELECT statement.

Although only the error values are of direct interest to developers who run a verification tool against their API calls, the other values can be used to judge the efficiency of a given verification tool's abstract interpretation techniques and the coverage of the approach. Given verification functions following the above conventions, any verification tool can invoke these functions using reflection or dynamic library loading techniques to verify API calls located in arbitrary libraries, for which the verification tool has no *a priori* knowledge.

The design we have described offers a number of advantages over incorporating the verification code directly in a tool.

- It places the burden of checking to the API developers: the ones who should best know how their API is to be used.
- It allows the checking code to interoperate with internal API functions, because the two can potentially live in the same C++/C# namespace or Java package space. This allows, for example, the checking code to invoke a DSL parser.
- Checking code is easier to write and contains fewer dependencies than *ad hoc* API verification code that comes together with a verification tool.
- Improvements of the checking tool, such as the incorporation of sophisticated abstract interpretation techniques, are automatically utilized by all existing API checking functions.
- The same checking code can be used by a number of different tools. Thus, the

significant effort required to write checks for the many thousands of API functions and methods does not need to be duplicated among different tools, while verification tools can still compete on the way they determine the arguments of an API invocation, efficiency, usability, and other features.

- The checking results across different APIs and methods can be readily summarized in a uniform manner to obtain an idea of the approach's coverage.

4 Implementation Examples

To validate the feasibility of our approach, we designed the application of our verification framework on Java methods, we added API verification functionality to the FindBugs tool (Hovemeyer and Pugh, 2004), and wrote verifiers for a small number of Java classes. Note that none of the above steps characterizes our approach. Our framework can be applied to different languages, can be integrated with other tools, and, of course, it can support a large number of API verification functions.

4.1 API Verification in Java

To apply the API verification approach to a particular language, one has to define the interface and calling conventions for the verification functions. For our Java implementation we defined these as follows.

- The verification methods are defined in a separate verification class, named by appending the letter `V` to the original class name.
- The verification classes are defined either in the same package as the actual API class, or in the package of the actual API class with the sequence `gr.aueb.dmst.apiv` prepended to it. This convention allows both a package's author and third parties to write API verification methods.
- Each verification method has as its first argument an array of `boolean` values. Each element of the array is true if the value of a corresponding subsequent argument could be statically determined at compile time.
- The second argument of each verification method is the original method's invocation target.
- Subsequent arguments of each verification method are of the same type as those of the corresponding API method.
- Verification methods shall return values from the enumeration `gr.aueb.dmst.apiv.VerifyResult` following the semantics we defined in Section 3.

4.2 *Extending FindBugs*

The second step for implementing an API verification system is to have a tool scan the source or the object code of a program, locate calls to API functions, statically evaluate their arguments, call the corresponding API verification function, and display the verification results. In our implementation we extended the FindBugs tool (version 0.9.2-dev). Specifically, we added to FindBugs's suite of bug detectors one that calls the API verification functions.

FindBugs works by examining the compiled Java virtual machine bytecodes of the programs it checks, using the bytecode engineering library (BCEL) (Dahm, 1999). It supports plug-in bug detectors, it performs a (rudimentary at the time of writing) abstract interpretation of the Java virtual machine instructions, and it has an extensive mechanism for reporting errors, both through a GUI and by textual output. FindBugs was therefore ideally suited as a platform for adding API verification functionality.

Our API verifier taps into the bytecode's method invocation point. It obtains the method's signature (its class, name, package, and argument types), which uniquely determines the called method. Then, by using Java's dynamic class loading and reflection capabilities, it tries to load and execute an API verification function, and transform the returned value into an appropriate FindBugs error message. FindBugs annotates each error message with the corresponding source file name and line number, and also contains functionality for localizing the messages for different languages.

Given the many services that FindBugs already provides, the implementation effort for the API verifier was modest, totaling about 500 lines of Java code. We therefore believe that adding API verification to other verification tools, compilers, or IDEs will be a similarly lightweight task.

4.3 *Class Verifiers*

To bootstrap and validate our approach we implemented a number of API verifiers; in the future we hope that libraries will come packaged with their verification code, in the same way as nowadays they include in their distribution their on-line documentation. We wrote about 100 method verifiers, spanning 12 different classes: `java.lang.Math`, `java.lang.String`, `java.sql.Connection`, `java.sql.Statement`, `java.util.regex.Matcher`, `java.util.regex.Pattern`, `org.apache.xpath.XPath`, `org.apache.xpath.CachedXPath`, `org.apache.xpath.XPathAPI`, `javax.xml.xpath.XPath`, `org.dom4j.Node`, and `org.dom4j.Document`. We chose the classes for a variety of reasons: `Math`, because it contained a number of functions with well-defined argument domain val-

ues; `String`, because we found it to be one of the classes with the highest number of afferent couplings among the projects we examined; and the rest to experiment with different DSL verification strategies.

Writing a method verifier is not difficult. In total, our verifiers, excluding the 3000-line ANTLR-based implementation of an existing ANSI SQL parser, consist of 1350 lines of Java code; about 14 lines for each verification method. We found ourselves employing three different approaches for implementing the verifiers. One involves directly checking the arguments for validity, as in the following verifier for the method `static double Math.log(double x)`.

```
public static VerifyResult.Value
log(boolean isConstant[], Math t, double x) {
    if (isConstant[1])
        if (x < 0)
            return VerifyResult.Value.DOMAIN_ERROR;
        else
            return VerifyResult.Value.CORRECT;
    else
        return VerifyResult.Value.UNVERIFIED;
}
```

The second approach involves actually executing the function with the known values and appropriate stubs, and catching any generated exceptions. This is the approach we used for implementing the constructor `String(byte[] bytes, String charsetName)`.

```
public VerifyResult.Value String(boolean isConstant[],
String t, byte[] bytes, String charsetName) {
    if (isConstant[2]) {
        try {
            String x = new String(new byte[] {}, charsetName);
        } catch (UnsupportedEncodingException e) {
            return VerifyResult.Value.NAME_ERROR;
        }
        return VerifyResult.Value.CORRECT;
    } else
        return VerifyResult.Value.UNVERIFIED;
}
```

Finally, the third approach involves implementing one verifier in terms of another. Needless to say, this approach results in very terse code; a clear boon when a class contains tens of similar methods. Here is as an example the verifier for the method `int executeUpdate(String sql, int[] columnIndexes)`.

```
public VerifyResult.Value executeUpdate(boolean isConst[],
Statement s, String sql, int[] columnIndexes) {
```

```
    return execute(isConst, s, sql);  
}
```

Although the class coverage of our verifiers is by no means representative, it is instructive to see the distribution of the returned error types. Of the verifiers 57 can return an argument error, 13 return a syntax error, 7 a domain error, and 3 a name error. Another 20 verifiers are implemented in terms of others, and thus return indirectly one of the above errors.

5 Empirical Evaluation

We ran our API verifier on the compiled code (application-specific and accompanying libraries) of the eight packages listed in Table 3. The Java archives we checked comprised 353MB (about 13 MLOC), and FindBugs invoking only our API verifier took 68 minutes on dual-CPU 2.2MHz AMD Opteron computer running the Java HotSpot 64-bit server 1.5.0 virtual machine. Thus, the bytecode throughput of the API verification was about 86kB/s—comparable to that of a Java compiler running on the same hardware (22kB/s). Therefore, the API verification could in the future conceivably be part of the compilation process. Our figures also allow us to extrapolate an approximate source code throughput of 3,000kLOC/s; this indicates that an IDE could perform API verification while a program is being edited.

From a total of 6.9 million method invocations, our 100 method verifiers matched 55 thousand invocations and could perform some argument checking on 25 thousand of them. The verifiers identified 800 potential errors, though, as we will see, many of them were false positives. In general, our approach is neither sound nor complete: it will identify as erroneous method invocations that are obviously correct, and it will also fail to detect errors that could be detected. However, the quality of the results depends to a large degree on the tool that applies the method verifiers. Therefore, once the investment in method verifiers is made, improvements to the analysis tools will increase the return on that investment.

The API verification methods that were called and reported errors, the extend to which they could verify their arguments, and the errors they detected, appear in Table 4. All the applications we verified are stable and of production-quality, therefore one should not expect to locate many errors remaining in them. We encountered two of the four possible error categories our framework defines: argument errors and syntax errors. Our application sample contained relatively few calls to mathematical functions, and from them even fewer could have their arguments checked; this explains the absence of domain errors. Furthermore, none of the three `String` methods that could return a name error got called; therefore, no name errors appear in our results.

Project name	Method invocations											
	Number of .jar files		Overall		Checked		Verification extent			Errors		
	Total	Unique	Total	Unique	Total	Unique	None	Part	Args	Full	Arg	Syntax
Centraview 2.0.6	124	360727	49671	2588	26	1330	2	1121	117	17	1	1
Compiere 2.5.2d	86	610633	104987	6620	34	2937	10	3471	124	51	27	27
Eclipse 3.1	230	1527312	364836	8001	20	4623	2	3032	225	119	0	0
Jahia 4.1.0	166	959215	154268	9984	36	5773	16	3758	163	75	199	199
Java 1.5.0	1	441159	75233	3505	25	1964	7	1364	145	25	0	0
JBoss 4.0.2	231	1642235	286944	11603	34	6518	8	4646	300	110	21	21
Netbeans 4.1	286	1099832	234060	9295	28	4296	6	3222	1661	92	18	18
OpenWFE 1.5.4	70	311954	56966	3917	25	2180	4	1582	101	35	15	15
Total		6953067		55513		29621	55	22196	2836	524	281	281

Table 3. API verification results for different Java projects

Method	Verification extent				Errors	
	None	Part	Args	Full	Arg	Syntax
String.String(char[],int,int)	2506	55	0	0	13	0
String.charAt(int)	6552	0	9403	1606	122	0
String.substring(int)	5285	0	5915	130	108	0
String.substring(int,int)	11990	0	6125	117	281	0
Connection.prepareStatement(String)	625	0	375	0	0	188
Statement.execute(String)	169	0	3	0	0	26
Statement.executeQuery(String)	496	0	47	0	0	63
Statement.executeUpdate(String)	235	0	0	0	0	4

Table 4

API verification error reports

Lacking an intimate knowledge of the verified code base, we could not judge all the results we obtained. Nevertheless, by opportunistically examining the subset of results that could be easily analyzed we identified some actual syntax errors stemming out of non-portable DSL constructs, and four classes of false positives.

5.1 Portability Problems

We saw in Section 1.2 that some DSL syntax errors might in fact be portability problems: constructs that might work correctly in one specific DSL dialect, but fail on a different one. As an example, the following SQL construct, part of the HSQLDB test data, fails to embed the nested `SELECT` statement in brackets.

```
sStatement.execute(
    "UPDATE Invoice SET Total=SELECT SUM(Cost*"
    + "Quantity) FROM Item WHERE InvoiceID=Invoice.ID");
```

Although HSQLDB will accept this construct in its current form, a different database implementation—for example Microsoft Access in this case—might flag it as an error. If an application provides the flexibility of running with different back-end database engines, such an error might manifest itself on a subset of the product’s installations: a problem that would be difficult to isolate.

A different class of portability problems might arise from successive extensions made to an API. Again, these might not be uncovered during development, but surface when the application is deployed to different clients. As an example, consider a call to `java.util.regex.Pattern.quote`. This method is a feature of Java 1.5, yet Sun’s compiler will happily accept it, even if the application is compiled for

a Java 1.4 environment. Our API verification framework can be easily extended to flag such invocations as errors. Conveniently, the corresponding checks can probably be automatically implemented by scanning JavaDoc comments for instances of the `@since` tag.

5.2 Control Graph Inference

Currently FindBugs analyzes a program's bytecodes without making any inferences regarding the flow of control. As a result, we identified a number of false positives that would be avoided by a tool invoking our API checking framework after performing some deeper inference analysis of the code's control paths. The following code fragment from the Eclipse source code is a typical example.

```
int numberOfDots = 0;
[...]
if ((numberOfDots % 2) == 0) return true;
if (numberOfDots == 1) return false;
if (tagName.charAt(lastDot - 1) == '0' &&
    tagName.charAt(lastDot - 2) == '.') return true;
```

FindBugs in conjunction with our checking framework will report that `charAt` can be called erroneously with a value of `-1` or `-2`. This could indeed happen by taking into account the initial value of `numberOfDots`, and that the code that follows only increments it. However, the first two `if` statements ensure that `numberOfDots` ≤ 2 , control will never reach the third `if` statement, and therefore the suspect `charAt` invocations.

5.3 API Postconditions

Performing sophisticated control path analysis of the program's code would eliminate a number of false positives. Even more could be eliminated by having the API checking framework incorporate additional knowledge regarding the properties of API calls. Consider the following code fragment from Apache Ant.

```
String replace(String data, String from, String to) {
    StringBuffer buf = new StringBuffer(data.length());
    int pos = -1;
    int i = 0;
    while ((pos = data.indexOf(from, i)) != -1) {
        buf.append(data.substring(i, pos)).append(to);
        i = pos + from.length();
    }
    buf.append(data.substring(i));
}
```

```

    return buf.toString();
}

```

Here our framework erroneously reported as an error a call of `substring(int, int)` with `pos` being negative. Analysis of the program's control flow would establish that when `substring` is invoked `pos` $\neq -1$, but in the above case `substring` would also fail for `pos < i`. To eliminate this possibility the analysis framework would have to know the range of the `indexOf` method:

$$\text{indexOf}(a, b) \in \{-1, b \dots \text{Integer.MAX_VALUE}\}$$

and that therefore in our case `pos` $\geq i$.

5.4 Performance Bottlenecks

In the following excerpt from Apache Tomcat our API verifier complains that in the last line `substring` can be called with the value of the second argument less than the first.

```

while ((pos = value.indexOf("$", prev)) >= 0) {
    if (pos == (value.length() - 1)) {
        sb.append('$');
        prev = pos + 1;
    } else if (value.charAt(pos + 1) != '{') {
        sb.append('$');
        prev = pos + 1; // XXX
    } else {
        int endName = value.indexOf('}', pos);
        if (endName < 0) {
            [...]
        }
        String n = value.substring(pos + 2, endName);
    }
}

```

Examining the code reveals that the reported error is, once again, a false positive, because the preceding calls to `indexOf` and `charAt` ensure that `value` will start at `pos` with the sequence `$`{, and therefore `endName` will be at least `pos + 2`. On the other hand, the report illustrates a slight case of inefficiency: the search for the closing brace could well start at `pos + 2`, as follows.

```

    int endName = value.indexOf('}', pos + 2);

```

Again, as program functionality increasingly moves to APIs, so do the performance bottlenecks. Locating and reporting problems, such as the above (and worse), would be a valuable extension to our approach.

5.5 Bugs in the FindBugs Program

As we wrote earlier, the version of FindBugs we used did not perform control flow analysis. Much to our surprise, by examining a number of mistakenly reported errors, we discovered that it also arrived on completely unwarranted values for some of the arguments. For example, in the following excerpt from Eclipse, FindBugs established that `arg` had the value of `-eclipseTask`.

```
String arg = getArgument(commands, "-eclipseTask");  
[...]  
String className= arg.substring(index + 1);
```

Although we did correct some argument type inference errors in FindBugs by submitting corresponding patches, it is clear that our API verification framework stretches FindBugs beyond the quality limits of its current implementation. Our verifier guards against such wrong inferences, but clearly more work is needed in this area.

6 Discussion

The implementation of our API verification framework, and its application on real-world code, taught us a number of valuable lessons. Some apply to our framework in general, while others are associated with FindBugs, which we chose as our implementation platform.

The imperative code we used for expressing the API verification functionality proved to be efficient in terms of code size and performance, reliable, and easy to apply. In Section 4.3 we wrote that implementing a verification method takes on average 14 lines of Java code. Learning to write verification code proved remarkably easy: this paper's second author wrote about half the verification functions, having as a guide only the source code of the existing ones. The compactness of the verification methods also contributed to their reliability: most worked on the first try. In contrast, while testing our methods we found that the existing regular expression verification functionality provided by FindBugs (`BadSyntaxForRegularExpression`), being written at a lower level of abstraction, contained a number of tests for the `replace` methods that were incorrect.

One could argue that developers, who often find it difficult to write down correct specifications, are unlikely to write down code to verify those specifications. Furthermore, one could say that developers could, out of laziness, short-circuit the verification code by always reporting that the corresponding API calls are correct. Although both concerns are valid, we believe that because our approach uses the

same expression medium as that of the actual API implementation, namely imperative code constructs, developers will feel familiar with it, and will therefore embrace it. In the context of the Java development community we are already witnessing two similar cases where developers increasingly embed elements into Java programs to enhance their code's non-functional properties: *comprehensibility*, through JavaDoc comments, and *testability*, through JUnit test cases (Spinellis, 2006).

Intriguingly, tests of our verification functions also demonstrated that the API documentation of widely used interfaces sometimes is incomplete or wrong. For example, we found that an invalid `flags` argument in Sun's regular expression parsing code will not throw an `IllegalArgumentException` exception, despite what the documentation claims. On the other hand, the `String.replaceAll` method can throw a `NullPointerException` exception, but does not document it, while the list of error conditions for the `String.copyValueOf` method appears to be incomplete. These discrepancies show that the exercise of adding verification code to an existing API, apart from aiding the reliability of the API's client code, can also add rigor to the API's documentation and implementation.

The choice of using FindBugs as the platform for implementing the API verification checker proved a mixed blessing. On the one hand it minimized the code we needed to write, thus demonstrating that once API verification classes get written and shipped with the corresponding API implementation, the verification of API calls can be easily integrated into a number of tools. The integration with FindBugs also provided us with an easy-to-use GUI, and allowed our approach to capitalize on the familiarity of the many programming groups that already use FindBugs as part of their build procedure. On the other hand, the inference engine of FindBugs left a lot to be desired, leading to a large number of false positives, possibly obscuring some real errors. The internationalized FindBugs interface also prompted us to adopt a design where the API verification methods return a result from a fixed number of errors. This allows the localization of the FindBugs interface, but limits the expressiveness and flexibility of a verification method's error report.

While writing the XPath verification methods we came across another problem of our approach. Some class hierarchies are rooted at an interface specification, and implement the interface in a number of different, but compatible, ways. In such cases, we had to implement essentially similar method verifiers for each class implementing the interface. This requirement is a limitation of Java's reflection design (Arnold et al., 2005); the only way to mitigate this problem within our framework is to implement the verification methods at the interface level, and have multiple verification method stubs forward the calls to the higher level actual verification methods.

7 Conclusions and Further Work

Our API verification framework is clearly complementary to other existing code verification approaches, such as runtime checks. Our approach can be integrated in the compilation cycle to catch some bugs early on. Runtime checks can potentially catch a wider range of errors, but they can be performed slightly later in the development cycle: at the earliest during unit testing. Because our approach does not depend on a specific tool, and it allows verification code to be embedded with a library's binary implementation, we hope that API implementers (or even third parties) will gradually build a large set of verification methods that different verification tools can tap.

On the implementation front, our approach can be extended with the addition of verification methods for many more classes. We hope that the availability of our implementation as open source software¹ will stimulate such an effort. In addition, our framework's efficacy can be enhanced by increasing the accuracy of FindBugs's abstract interpretation of Java bytecodes, or by integrating API verification with a Java compiler, or a verification tool incorporating a theorem prover. Implementing the API verification framework for other languages, such as C and C#, and other APIs is also another worthwhile pursuit.

Our API verification framework also opens a number of interesting research questions. The representation of API call postconditions within the binary code of a library is an interesting problem. Our current design allows verification methods to check for preconditions. However, as we described in Section 5.3, the code driving the verification methods could provide them with more accurate argument values, if previously called verification methods could indicate the postconditions (e.g. their return value) for the arguments they received.

Furthermore, one would like to be able to verify stateful interactions of API calls—for example the double locking of a resource—again without incorporating API-specific knowledge within the verifier. Such extensions could also allow the API verifier to identify higher-level problems associated with the application's performance. Finally, given the availability of the API verification functions, one could envisage them being used to check an application's behavior at runtime (Pnueli et al., 2006). Under such a scheme, Aspect-oriented technology (Kiczales et al., 2001) could associate API calls with code that would first call the verification function (for a specific set of arguments) and then the actual API function. Such an approach would be especially useful for APIs whose functions can silently fail, such as the Windows Win32 API and the Single UNIX Specification.

¹ <http://www.dmst.aueb.gr/dds/sw/api-verify>.

8 Acknowledgments

We would like to thank the authors of FindBugs for the work they put into the platform, and in particular Dave Brosius for his help in integrating our type-inference patches that made it possible to implement our tool. We also thank the paper’s anonymous referees for many detailed and perceptive comments.

References

- Arnold, K., Gosling, J., Holmes, D., 2005. *The Java Programming Language*, 4th Edition. Addison-Wesley, Boston, MA.
- Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., Washington, R., 2005. Combining test case generation and runtime verification. *Theoretical Comput. Sci.* 336 (2-3), 209–234.
- Ball, T., Cook, B., Levin, V., Rajamani, S. K., 2004. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. Tech. Rep. MSR-TR-2004-08, Microsoft Research, Redmond, WA.
- Ball, T., Rajamani, S. K., 2002. 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*. pp. 1–3.
- Barach, D. R., Taenzer, D. H., Wells, R. E., 1982. A technique for finding storage allocation errors in C-language programs. *SIGPLAN Notices* 17 (7), 32–38.
- Barringer, H., Finkbeiner, B., Gurevich, Y., Sipma, H. B. (Eds.), 2006. *Proceedings of the Fifth Workshop on Runtime Verification (RV 2005)*. *Electronic Notes in Theoretical Computer Science*. 144(4).
- Beck, K., Gamma, E., 1998. Test infected: Programmers love writing tests. *Java Report* 3 (7), 37–50.
- Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2003. A static analyzer for large safety-critical software. In: *PLDI ’03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. pp. 196–207.
- Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., Poll, E., Jun. 2005. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer* 7 (3), 212–232.
- Bush, W. R., Pincus, J. D., Sielaff, D. J., 2000. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience* 30 (7), 775–802.
- Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H., 2003. Modular verification of software components in C. In: *ICSE ’03: Proceedings of the 25th International Conference on Software Engineering*. pp. 385–395.
- Chen, H., Wagner, D., 2002. MOPS: An infrastructure for examining security prop-

- erties of software. In: CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security. pp. 235–244.
- Cok, D. R., Kiniry, J. R., 2005. ESC/Java2: Uniting ESC/Java and JML — progress and issues in building and using ESC/Java2. In: Barthe, G., Burdy, L., Huisman, M., et al. (Eds.), Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004. Springer-Verlag, pp. 108–129, Lecture Notes in Computer Science 3362.
- Copeland, T., 2005. PMD Applied. Centennial Books.
- Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, Zheng, H., 2000. Bandera: Extracting finite-state models from Java source code. In: ICSE '00: Proceedings of the 22nd International Conference on Software engineering. pp. 439–448.
- Cousot, P., Cousot, R., 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252.
- Csallner, C., Smaragdakis, Y., 2005. Check 'n' crash: Combining static checking and testing. In: ICSE '05: Proceedings of the 27th International Conference on Software Engineering. pp. 422–431.
- Dahm, M., 1999. Byte code engineering. In: Cap, C. H. (Ed.), JIT '99, Java-Informationen-Tage 1999. Springer-Verlag, pp. 267–277.
- Das, M., Lerner, S., Seigle, M., 2002. ESP: Path-sensitive program verification in polynomial time. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. pp. 57–68.
- DeLine, R., Fähndrich, M., 2001. Enforcing high-level protocols in low-level software. In: PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 59–69.
- DeLine, R., Fähndrich, M., 2004. The Fugue protocol checker: Is your software Baroque? Tech. Rep. MSR-TR-2004-07, Microsoft Research, Redmond, WA.
- Engler, D., Chen, D. Y., Hallem, S., Chou, A., Chelf, B., 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In: SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. pp. 57–72.
- Evans, D., Guttag, J., Horning, J., Tan, Y. M., 1994. LCLint: A tool for using specifications to check code. In: SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering. pp. 87–96.
- Evans, D., Larochelle, D., Jan./Feb. 2002. Improving security using extensible lightweight static analysis. IEEE Software 19 (1), 42–51.
- Fagan, M. E., 1976. Design and code inspections to reduce errors in program development. IBM Syst. J. 15 (3), 182–211.
- Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M., Mar. 2002. DrScheme: A programming environment for Scheme. Journal of Functional Programming 12 (2), 159–182.
- Flanagan, C., Leino, K. R. M., 2001. Houdini, an annotation assistant for ESC/Java. In: FME '01: Proceedings of the International Symposium of Formal Methods

- Europe on Formal Methods for Increasing Software Productivity. pp. 500–517.
- Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., Stata, R., 2002. Extended static checking for Java. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. pp. 234–245.
- Guttag, J. V., Horning, J. J. (Eds.), 1993. Larch: Languages and Tools for Formal Specification. Springer-Verlag.
- Hallem, S., Chelf, B., Xie, Y., Engler, D., 2002. 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. pp. 69–82.
- Havelund, K., Roşu, G., 2004. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design* 24 (2), 189–215.
- Hovemeyer, D., Pugh, W., 2004. Finding bugs is easy. In: OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. pp. 132–136.
- Jackson, D., 1995. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering Methodology* 4 (2), 109–145.
- Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y., 2002. Cyclone: A safe dialect of C. In: USENIX Annual Technical Conference. pp. 275–288.
- JLint, 2004. JLint. Available online <http://jlint.sourceforge.net/>.
- Johnson, S., 1977. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ.
- Johnson, S. C., Lesk, M. E., Jul.-Aug. 1987. Language development tools. *Bell System Technical Journal* 56 (6), 2155–2176.
- Jones, C., 1990. Systematic Software Development using VDM. Prentice Hall, Englewood Cliffs, NJ.
- Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W., 2001. Getting started with ASPECTJ. *Commun. ACM* 44 (10), 59–65.
- Larus, J. R., Ball, T., Das, M., DeLine, R., Fähndrich, M., Pincus, J., Rajamani, S. K., Venkatapathy, R., May/June. 2004. Righting software. *IEEE Software* 21 (3), 92–100.
- Leavens, G. T., Baker, A. T., Ruby, C., 2005. Preliminary design of JML: A behavioral interface specification for Java. Tech. Rep. TR #98-06-rev28, Department of Computer Science, University of Iowa, Ames, IA.
- Lindahl, T., Sagonas, K., 2004. Detecting software defects in telecom applications through lightweight static analysis: A war story. In: APLAS 2004: Second Asian Symposium on Programming Languages and Systems. Springer, pp. 91–106, Lecture Notes in Computer Science 3302.
- Mandelin, D., Xu, L., Bodík, R., Kimelman, D., 2005. Jungloid mining: helping to navigate the API jungle. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 48–61.
- Marinov, D., Khurshid, S., 2001. TestEra: A novel framework for automated testing of Java programs. In: ASE '01: Proceedings of the 16th IEEE International Con-

- ference on Automated Software Engineering. IEEE Computer Society, Washington, DC, USA, p. 22.
- Meyer, B., 1997. *Object-Oriented Software Construction*, 2nd Edition. Prentice Hall PTR, Upper Saddle River, NJ.
- Pnueli, A., Zaks, A., Zuck, L. D., 2006. Monitoring interfaces for faults. *Electronic Notes in Theoretical Computer Science* 144 (4), 73–89.
- Reimer, D., Schonberg, E., Srinivas, K., Srinivasan, H., Alpern, B., Johnson, R. D., Kershenbaum, A., Koved, L., 2004. SABER: Smart analysis based error reduction. In: *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. pp. 243–251.
- Rutar, N., Almazan, C. B., Foster, J. S., 2004. A comparison of bug finding tools for Java. In: *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE '04)*. pp. 245–256.
- Spinellis, D., Feb. 2001. Notable design patterns for domain specific languages. *Journal of Systems and Software* 56 (1), 91–99.
- Spinellis, D., 2006. *Code Quality: The Open Source Perspective*. Addison-Wesley, Boston, MA.
- Szyperski, C., 2002. *Component Software: Behind Object-Oriented Programming*, 2nd Edition. Addison-Wesley, Reading, MA.
- Viega, J., Bloch, J. T., Kohno, Y., McGraw, G., 2000. ITS4: A static vulnerability scanner for C and C++ code. In: *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*. p. 257.
- Weiser, M., 1981. Program slicing. In: *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*. pp. 439–449.
- Wright, A. K., Cartwright, R., Jan. 1997. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst.* 19 (1), 87–152.

Id: api-verify.tex,v 1.45 2006/09/17 19:50:45 dds Exp