Tracking Down Software Bugs Using Automatic Anomaly Detection

Sudheendra Hangal
Sun Microsystems India Pvt. Ltd.
Divyasree Chambers, Shantinagar
Bangalore 560025
sudheendra.hangal@sun.com

Monica S. Lam
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
lam@cs.stanford.edu

ABSTRACT

This paper introduces DIDUCE, a practical and effective tool that aids programmers in detecting complex program errors and identifying their root causes. By instrumenting a program and observing its behavior as it runs, DIDUCE dynamically formulates hypotheses of invariants obeyed by the program. DIDUCE hypothesizes the strictest invariants at the beginning, and gradually relaxes the hypothesis as violations are detected to allow for new behavior. The violations reported help users to catch software bugs as soon as they occur. They also give programmers new visibility into the behavior of the programs such as identifying rare corner cases in the program logic or even locating hidden errors that corrupt the program's results.

We implemented the DIDUCE system for Java programs and applied it to four programs of significant size and complexity. DIDUCE succeeded in identifying the root causes of programming errors in each of the programs quickly and automatically. In particular, DIDUCE is effective in isolating a timing-dependent bug in a released JSSE (Java Secure Socket Extension) library, which would have taken an experienced programmer days to find. Our experience suggests that detecting and checking program invariants dynamically is a simple and effective methodology for debugging many different kinds of program errors across a wide variety of application domains.

1. INTRODUCTION

While rapid advances in computing hardware have led to powerful, multi-gigahertz processors, advances in software reliability have not kept pace with this progress. Software program bugs continue to be frequent, in spite of increasing requirements that software be reliable. Non-stop systems have stringent uptime requirements and must be kept running even in the face of hardware or software errors and may be required to be monitored, debugged and patched on the fly. While software program crashes are problematic enough, perhaps more dangerous are undetected errors which silently compromise the results of a computation. For example, it is difficult to verify the results of a software simulation of a system, since the purpose of simulation is to predict the behavior of the system without having to build it. The use of incorrect intermediate results due to undetected bugs has been known to lead to catastrophes in mission-critical or even safety-critical situations [9][12]. All this calls for a much deeper understanding of what happens inside a software program than the conventional visibility offered by the outputs of a program.

The challenge of building reliable software is compounded in real life by the fact that programmers often do not take the time to write detailed specifications or documentation. As a result, software documentation is frequently incomplete or out of date. In addition, complex software systems are so large that one person rarely has knowledge about all parts of the system. Quite often, software systems are assembled using multiple components, which may have been developed by different groups of people, perhaps in different organizations, using different development and testing methodologies. Bugs in such systems, especially those which arise only in rare corner cases, can take days or weeks to debug. It is thus desirable to have automated debugging methods which utilize the vast power of machines available today to reduce human debugging time.

We tackle the problems of both detecting bugs and hunting down the root causes of bugs using the concept of dynamic invariant detection and checking. Most programs obey many invariants, many of which are not documented anywhere, and in fact, may not be known even to the original writers of the code. Explicitly specifying known program invariants, usually with the goal of documenting the programs, or of checking the invariants dynamically or statically [4][5][6] is a tedious task. In addition, manual specification of invariants tends to capture only a few abstract, high-level invariants and a few implementation-specific, low-level invariants at key program points, usually reflecting the parts of the program that programmers tend to think and worry about the most.

In contrast to static specification, we can automatically detect likely program invariants based on dynamic program behavior (also called "dynamic invariants" in this paper) [8]. The Daikon tool, developed by Ernst et al., detects dynamic invariants by starting with a specific space of possible program invariants. It runs the program on a large set of test inputs, and infers likely invariants by ruling out those which are not violated during any of the program runs. This approach has the advantage of being automatic and pervasive, unlike static specification, but is limited by the fixed set of invariants hypothesized and checked for. Dynamic invariant detection is also constrained by the quality of test inputs available.

This paper introduces a new tool we have developed, called DIDUCE (Dynamic Invariant Detection ∪ Checking Engine).

ICSE'02, May 19-25, 2002, Orlando, Florida, USA.

Like Daikon, DIDUCE tries to extract invariants dynamically from program executions. However, instead of presenting the user with numerous invariants found after a program's execution, DIDUCE continually checks the program's behavior against the invariants hypothesized up to that point in the program's run(s) and reports all detected violations. When a dynamic invariant violation is detected, the invariant is relaxed to allow for the new behavior and program execution is resumed. This results in a fully automatic tool that checks a program against a model it creates without requiring any human intervention. The ability to detect program anomalies has many interesting applications. For example, it allows the user to ask DIDUCE "what's new?" just before a program crashes, and the answer often points to the source of the error. Anomalies, or invariant relaxations, are ranked according to a statistical confidence level to help users quickly locate the largest deviations from previous behavior.

By focusing on the anomalies in a program's execution, DIDUCE provides uncluttered visibility into the noteworthy part of a program's behavior. Note that the deviations reported may not necessarily cause an error or result from an error. However, knowing certain execution scenarios to be rare may itself be insightful for the user. In practice, we found that users were genuinely interested in knowing what these corner cases were. They liked how DIDUCE gave them a better feel for what their program was doing.

In the presence of a software error, dynamic invariant violations may point to the first consequence of the bug or may identify the unique context under which the bug takes place. For example, DIDUCE may detect a change in the input pattern that subsequently triggers a bug. While DIDUCE cannot track down all possible bugs, its value lies in the fact that when it does so, it can often detect a bug at its source, which can be very far away from where the bug actually manifests itself (if indeed it ever does.) Even for users debugging program execution backwards from a point of failure, invariant violations often leave an interesting trail of anomalous events, which the users can work backwards through.

For a debugging tool to be useful, it must be efficient and easy to use. The tool must be able to handle large programs. As we shall see in the results of our experiments, many useful invariants are serendipitous in nature, and it is important that invariant detection be pervasive enough to catch all such violations.

The main contributions of this paper are as follows.

1. This paper introduces the concept of dynamic invariant detection and checking as a means to aid programmers in finding the root causes of software bugs. This technique is especially valuable for debugging complex algorithmic errors. The information is also useful in helping programmers understand their software better.
2. We have implemented this concept in a tool called DIDUCE that works with Java bytecodes. To handle large programs, DIDUCE keeps track of relatively simple program invariants. We have developed a succinct representation to capture important invariants. DIDUCE instrumentation is modular, allowing parts of the software to be instrumented, one at a time. While DIDUCE has reasonable defaults that make it easy for novices to use, it is also extensible, allowing sophisticated users to tailor it to a particular application. DIDUCE has a GUI interface which allows users to easily navigate through the invariant violations detected and correlate the violations with the source code, if it is available.
3. We have applied DIDUCE to four real-life Java programs of significant size and complexity, and the tool succeeded in identifying the root causes of programming errors in each of the programs quickly and automatically. In particular, we used DIDUCE to effectively isolate a timing-dependent bug in a released Java Secure Sockets Extension library which would have taken an experienced programmer more than a couple of days to find. Our experience suggests that detecting and checking program invariants dynamically is a simple and effective methodology for debugging many different kinds of program errors across a wide variety of application domains.

The rest of this paper is organized as follows. Section 2 describes some possible usage models for a tool like DIDUCE. Section 3 discusses what invariants can be tracked by DIDUCE, and their representation. Section 4 describes the implementation details of DIDUCE. Section 5 describes how we used DIDUCE to find bugs in four large Java programs. Section 6 compares DIDUCE with related work. Finally, Section 7 concludes and presents some ideas for future work.

2. Usage Models for DIDUCE

Dynamic invariant detection has been proposed in the past as a way to extract invariants automatically from programs [8]. These invariants help programmers understand the program and can also be fed to a static checker as hypotheses to be proved or disproved [15]. By turning this concept into an online system that also checks for violations, we have created a tool with wide applicability in the software development process. While this paper focuses on the use of this concept in debugging, it is also useful for program testing and is helpful in evolving software correctly, as described below.

Debugging programs that fail on some inputs: It is a common occurrence for a program which works correctly on many inputs, to fail on others. DIDUCE can be used to quickly pinpoint differences in behavior between the successful and the failing runs. For example, DIDUCE could be used to automatically provide debug information upon detection of regressions in a test suite, by first extracting invariants from test cases that pass, and checking for invariant violations on the failing cases. The list of invariant violations can then be presented along with the failing tests to a human test engineer or developer to reduce debugging time.

Debugging failures in long-running programs: Some of the hardest bugs to track down are those that occur only after a program has executed for a long time. Typically, a developer would guess what the problem is and try to gain visibility on the suspect variables or code segments by adding debugging statements, assertions, and breakpoints into the program. This trial-and-error process can be time consuming for long-running programs. Moreover, a developer's intuitions may not necessarily be dependable especially if the errors are caused by his own misconceptions in the first place. DIDUCE blindly and continually monitors all the variables in the program and is better suited to locating such errors. For long running programs, a training set may not even be necessary. Assuming there are no bugs in the early part of the run, DIDUCE can flag anomalous behavior in the later parts of the run.

Debugging component-based software: DIDUCE can be used in the bring-up of a component-based system. Normally, a program must run correctly on some inputs, or for some

duration, before DIDUCE can successfully extract meaningful invariants for the program. For component-based software, however, we can first train DIDUCE on other codes that use the same components correctly, and apply it to check the behavior of the component in the context of the new software.

Testing programs with inputs for which the correct outputs are unknown: Producing test cases for a program can be quite tedious because the expected results must also be prepared for comparison with the program's output. Assuming there are some tests for which the results are known, we first train DIDUCE on these tests, and use the invariants gathered to check the runs on inputs with no known outputs. With this approach, it is possible to test software with, say, pseudo-random inputs for which no answers are known a priori. Invariant violations detected indicate bugs in the program, or at least expose corner cases which did not occur during directed testing.

Assisting in program evolution: Another use for DIDUCE is to check if modifications to a part of the program unexpectedly alter the behavior of other parts. We can simply check the invariants collected before the update from the unmodified parts of the program on program runs after the update. This is especially useful in assisting new programmers, who may not understand all parts of the system, to ensure that changes they make do not break invariant assumptions in the rest of the software.

3. DIDUCE INVARIANTS

DIDUCE is an on-line invariant detector and checker for Java programs. The system instruments the user's program and maintains invariants on the values of a set of *tracked expressions* at various program points. For each tracked expression, DIDUCE maintains an invariant hypothesis that is satisfied by all the values that have occurred in the history of the execution so far. If an expression is found to evaluate a value that does not conform to its current invariant hypothesis, DIDUCE relaxes the invariant to allow for the new value.

DIDUCE operates in one of two modes: the training mode and the checking mode. The only difference between these two modes is that in the training mode, DIDUCE silently learns invariants by relaxing invariant hypotheses as needed; in checking mode, DIDUCE emits messages about invariant relaxations which occur along the way. Training continues in the checking mode as well. DIDUCE will therefore warn about a value which violates a presumed invariant only the first time that it is encountered; the invariant is thereafter relaxed to silently allow that value. Of course, code which gets executed for the first time is treated as a special kind of invariant violation.

The rest of this section describes which program points are instrumented, what expressions are tracked, how invariants are internally represented, and the notion of invariant confidence.

3.1 Instrumented Program Points

DIDUCE associates invariants with static program points, i.e. specific locations in the program's code. Instrumentation code is introduced at these program points to evaluate the values of a set of tracked expressions, to report any invariant violations, and to update the dynamic invariant according to the new value. DIDUCE instrumentation works directly on Java bytecodes and does not need access to source code.

DIDUCE allows tracked expressions to be attached to the following categories of program points:

1. program points which read from or write to objects (including arrays),
2. program points which read from or write to a static variable, and
3. procedure call sites

This design gives the user visibility into the global state of the computation, as captured by the contents of objects and procedure interfaces. We leave out stack accesses as they are time-consuming to track and are less interesting, especially since all Java objects are allocated on the heap.

Ordinarily the user supplies DIDUCE with a list of class files or JAR (Java Archive) files associated with the program, excluding the standard Java libraries. By default, DIDUCE will instrument all the static program points described above in those classes. This is the recommended mode of operation, especially for a user who knows very little about the program being debugged. DIDUCE also allows the user finer-grained control over the instrumentation. Users can restrict the instrumentation to certain classes, methods, fields accessed, types of accesses (read or write of a static field, or an object, or an array), line numbers (provided line number information is present in the class file), and specific parameters or return values in calls at specific call sites. This feature not only reduces run-time overhead by omitting instrumentation of program points of no interest, but also eliminates noise generated from uninteresting points in the program.

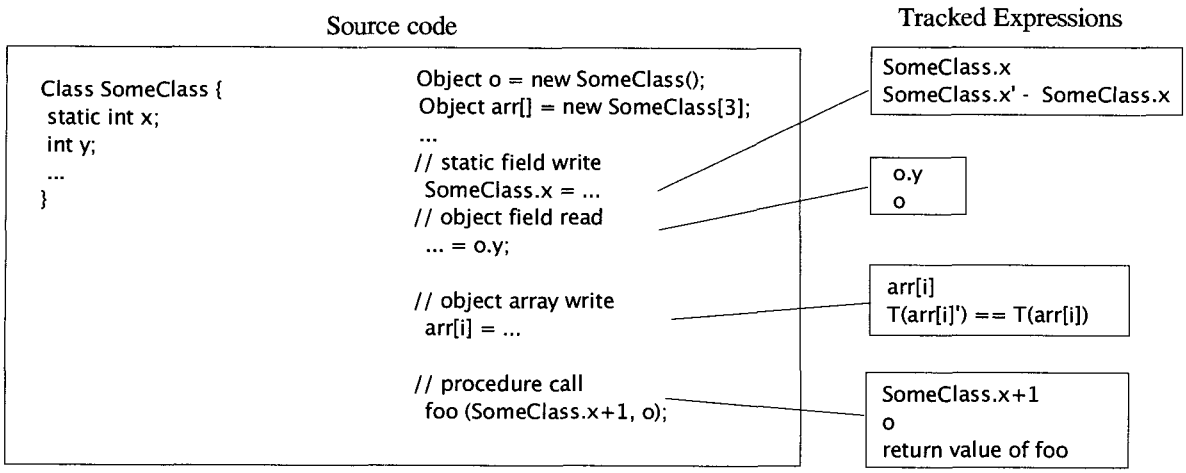
DIDUCE does not allow the user to qualify the tracked expressions dynamically. While such a feature may allow a user to, for example, track an expression only if executed by a particular thread or, under a particular calling context, this option would have increased the dynamic execution overhead per tracked expression. In contrast, our simple, static approach of selecting instrumentation points is time and space efficient. The storage overhead is proportional to the number of invariants tracked, which in turn is proportional to the static size of the program. The technique is modular in that each program point can be instrumented without regard to which other program points are chosen for instrumentation. Thus, parallelism can be used easily to speed up the analysis. We routinely generate different versions of the program being debugged by instrumenting different parts of the code, and run these versions in parallel on different machines.

3.2 Tracked Expressions

For each instrumented program point, DIDUCE derives invariants that are obeyed by all the objects/variables accessed at that program point. As discussed above, these are either (1) objects read or written, (2) static variables read or written, or (3) input parameters or return values, depending on the program point.

Specifically, DIDUCE associates with each program point a set of expressions, each of which is a function of the object or variable being accessed. DIDUCE maintains an invariant for each expression in this set, starting with the strictest invariant assumption at the beginning and gradually relaxing it to encompass the values observed for the expressions. The relaxation rules are based on the type of the expression, which can be one of boolean, byte, short, char, integer, long or reference. We currently ignore all values of floating point data types, since invariants on them tend not to be meaningful with our default representation. Details on the representation of the invariants and their relaxation rules are given in Section 3.3.

DIDUCE tracks the following expressions for each program point by default:



For writes, the variable name (e.g. o.x) refers to its value before the write, while the name with a ' suffix (e.g. o.x') refers to its value after the write. T(o) refers to the runtime type of object o.

Fig. 1: Some Examples of Tracked Expressions

- the value being read or written
- the parent object, in the case where a field of an object is accessed (except for arrays.)
- the difference between the values of the location accessed before and after a write operation. For data of a numeric type, this difference is simply their numeric difference. As a useful side-effect, this expression also tracks if the written field changes monotonically with respect to this program point. If the data is of reference type, the difference is a boolean value indicating whether the run-time type of the new value matches the old.

Fig. 1 illustrates a few examples of tracked invariants at different kinds of program points. For tracked expressions which are of reference type, it does not make sense to capture the value of the object; we therefore map objects to their run-time types instead. Invariant violations on such expressions try to catch scenarios where new run-time types are observed for that program point. Null values are treated as a special run-time type of their own.

While we consider the set of default invariants described above to be generally useful in practice, invariants can be specialized for a particular program or domain. Users can do so by extending one of the classes in the DIDUCE run-time library. Users can omit from the default set those invariants considered not useful for their application. For example, users may not want to track invariants on run-time types when there is no polymorphism in the program. Users may also specify their own set of expressions to be tracked for different categories of program points. The tracked expression has to be a function of the parent object being accessed (for object reads or writes), the value of the field being read or written, and (for writes) the old value of the location being written. For example, a tracked expression can use the parent object reference to compute properties like the array length at array access points, or it can compute a function of multiple fields in the object. Users can also specify that DIDUCE should treat invariants on the tracked expressions at different program points as belonging to the same program point. This is useful, for example, when users would like to maintain invariants on a particular property regardless of the location in the program code.

3.3 Invariant Representation

To keep the time and space overhead low, it is important that DIDUCE invariants be represented compactly and be easy to compute and update. For example, it would be too expensive to keep track of all the values seen for each tracked expression; nor would this necessarily be useful because over-learning may result in generating too many invariant violations.

For each instrumented program point, DIDUCE keeps track of the number of times that program point is executed, and maintains an invariant for each of the expressions tracked. We describe below the default procedure, which can also be over-ridden by the user if so desired.

First, DIDUCE reduces the values of all expressions, which may be of type boolean, byte, char, short, int, and long, to integers. Tracked expressions of reference type are mapped to an integer by computing the *hashcode* of the `String` object representing the name of the run-time type. Hashcode is a function in the standard Java library, which when applied to a string, will always return the same hash code across all runs of the program. This is important because we wish to carry over invariants derived from one program run to other runs. Null objects map to a hashcode value of 0. This mapping is relatively quick to compute, requiring 3-4 memory references on most systems.

For each expression, whose values are now all integers, the invariant maintains for each bit position (1) the value of that bit the first time the expression was evaluated, and (2) whether different values have been observed for that bit position. A violation is reported if differences between the new value and previous ones are observed in new bit positions.

Specifically, we associate with each expression a tuple of two integers, an initial value V and a mask M. The ith bit in M is set to 1 iff the same bit value has always been observed for that position. Suppose the first value of an expression is W, then

$M := \neg 0$, $V := W$.

Suppose, subsequently, the expression returns W', if

$(W' \otimes V) \wedge M \neq 0$,

where $\otimes$ is the "xor" operator, a violation is reported, and the invariant is relaxed by:

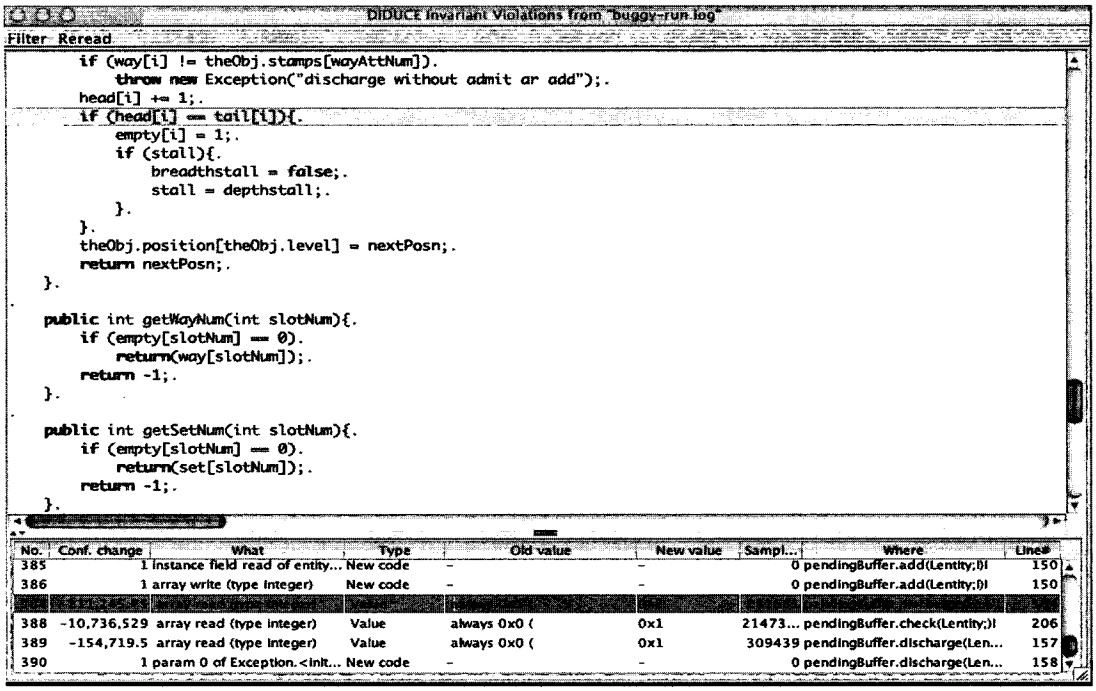


Fig. 2: Screenshot of DIDUCE GUI reporting invariant violations

$M := M \wedge \neg (W' \otimes V)$

Thus, checking and updating invariants for each tracked expression are efficient in both space and time.

This succinct representation keeps track of some interesting properties about the values observed for an expression. For example, it can tell if an expression evaluates to the same value all the time. For numerical expressions, the invariants keep track of (1) whether the values were only positive or only negative, only odd or only even, (2) an approximate upper bound on the value, and (3) which of the bits have constant values. To simplify implementation, we currently employ the same representation for expressions of reference type as well. DIDUCE can therefore detect whether the same run-time type is encountered at that program point all the time. In our experiences, we found that a good use of the reference type invariants was to distinguish between null and non-null values.

With this representation, the number of violations detected for each expression can be no greater than the number of bits in a word. It also has the desirable property that if the values of an expression increase monotonically by a constant amount (such as a counter counting upwards), the frequency of violations decreases exponentially.

In summary, the storage required for maintaining invariants is about three words per tracked expression (one to store the number of times the program point is executed and two to store the invariant). The run-time overhead is limited to a few memory operations and a few simple logical operations per instrumentation point.

3.4 Invariant confidence

We introduce the notion of *confidence* on invariant hypotheses to help users prioritize among invariant violations, which can be quite numerous, especially at the beginning of a program run. Roughly speaking, we have high confidence in an invariant hypothesis if the expression has been evaluated many times and there is little variation in the values observed. The confidence level of an invariant is therefore defined as the ratio between the number of times the expression has been evaluated and the number of values the invariant accepts. The number of values accepted is simply 2^n, where n is the number of 0s in the mask vector of the corresponding invariant. This simple metric works quite well from our experience, as it strongly biases the confidence towards expressions which take on only one value, or a few values which are close together.

Every invariant violation is reported with the change in confidence levels between the old invariant and the newly relaxed invariant. A large drop in confidence signals a noteworthy invariant violation. Since this measure of change in confidence is not meaningful in the case of invariants on run-time types (which are highly sensitive to the type names and hashcode encoding scheme used), we report a fixed, user-specifiable confidence level whenever a violation on a run-time type is detected. Similarly, code executed for the first time is reported with a fixed, user-specifiable invariant confidence change.

4. DIDUCE IMPLEMENTATION

Our goal is to make the DIDUCE system as easy to use as possible. Using DIDUCE only involves inserting two additional steps before running the program. The user first specifies a list of class files, optionally with a specification of which categories of program points to instrument, as described in Section 3.1. The instrumented class files go into a DIDUCE JAR file; the user inserts this JAR file at the head of the classpath and then runs the program as usual. Internally, in the instrumentation step, we use the ByteCode Engineering Library (BCEL) [3] to instrument Java class files and insert calls to the DIDUCE run time system at appropriate program points.

Since DIDUCE instruments legal and verifiable Java class files, source code is not required at the instrumentation step. Adding instrumentation to a class file can potentially cause the class to exceed static class file limits [13]; however, we have not

Program name	Description	# Lines of Source Code	# Classes (instrumented/total)	# Instrumented program points	Slowdown factor
Simulator	Proprietary performance simulator for multiprocessor memory systems	3300	10/28	3204	8-12X (Using 10 machines)
Mailmanage	Open source mail management utility	1700 (+ ~ 20000 JavaMail library)	214/214 (203 classes in JavaMail library)	13014	6X
JSSE Library	Shipping reference implementation for Java Secure Sockets Layer Library	30000 (+ Obfuscated RSA libraries)	384/384	34844	8X
Joeq	Research project to develop a Java Virtual Machine	31500	18/137	3371	20X

Table 1: Details of programs DIDUCE was tried on

found this to be a problem on any real-life programs. Apart from this possibility, the instrumented versions of the classes generated by DIDUCE are completely legal and verifiable Java class files and can be run on any compliant Java Virtual Machine. While access to source code is not required for instrumentation, it is, of course, useful for understanding the invariant violations reported.

To allow aggregation of invariants across different program runs, users can optionally save the set of learned invariants to a file, and initialize invariants at the beginning of a run from the saved file. Users can also specify whether DIDUCE is to be run in training mode or checking mode. As stated earlier, the only difference between these modes is that DIDUCE emits invariant relaxation messages in the checking mode.

As the instrumented program runs, users can tell DIDUCE to report invariant violations only if they are above a minimum level of confidence change. The results about invariant violations are fed back in real time to a GUI (See Fig. 2) which allows the user to filter invariants, sort them by confidence change, and browse through associated source code, if it is available. Users can also browse through detected invariants at the end of a run.

DIDUCE invariant detection and checking is safe in a multithreaded environment. An online monitoring system like DIDUCE is very useful for multithreaded programs which are otherwise difficult to debug. Of course, since instrumenting the program may perturb the program's timing characteristics, DIDUCE is only useful in these scenarios if the instrumented program still exhibits the bug.

Thus far, we have concentrated mainly on DIDUCE functionality, and have not put in much effort towards reducing the run-time overhead. An instrumented program using the default settings currently runs one to two orders of magnitude slower. Note that the run-time instrumentation overhead only affects the computation part of the program. DIDUCE has no overhead on native network or I/O operations. Thus applications which include a significant amount of network or I/O activity may see relatively smaller overheads. As mentioned before, it is possible to make parallel runs of the program, with different parts instrumented.

5. DIDUCE EXPERIENCES

To evaluate the effectiveness of the DIDUCE system, we applied it to four significantly complex Java projects. The first program, Simulator, is a proprietary timing-accurate simulator for a class of sophisticated memory systems being considered for a multiprocessor-on-a-chip implementation of the MAJC architecture [19]. The second, Mailmanage, is an open-source email management utility, developed by a team including one of the authors of this paper. The third is the JSSE (Java Secure Sockets Extension) code, which has been a standard extension to the Java library for over one and a half years. Finally, the fourth program, Joeq, is a Java Virtual Machine system developed by one of the graduate students in our group. Both the Mailmanage and Joeq projects are available at the open source web site, SourceForge. On each of these programs, we only specified the list of classes to instrument - all other customizable parameters such as the set of instrumented program points, tracked expressions and invariant representation were set to the defaults described in Section 3. Table 1 summarizes relevant parameters for each program. We chose these programs for our experiments because they happened to be real programs that we came across, as we were developing DIDUCE. We have not yet tried to use DIDUCE extensively on other programs.

In all the four examples, we found (not surprisingly) that DIDUCE was especially helpful in pinpointing late-stage bugs that occur after many test cases are already running. Late-stage bugs are usually the hardest to find and take the longest to analyze. Furthermore, no up-front investment needed to be made by programmers in terms of specifying invariants - they started using the tool only when they needed to debug their programs.

5.1 MAJC Memory System Simulator

MAJC is a CPU architecture developed at Sun with support for on-chip multiprocessing. To model future implementations of the architecture, the processor designers were using a simulator to evaluate various memory system designs. At the time we applied DIDUCE to the simulator, the program was almost fully developed and was deemed fairly stable, so much so that its results were already being used to make architectural design decisions. Since this was a simulator meant only to estimate performance, it ran through pre-generated program traces without actually executing the simulated program. The simulator's output was the number of clock cycles taken to execute the entire trace. Since there was no obvious way to verify that its results were correct, the programmer had laced it liberally with assertion checks throughout the code.

We instrumented each of the ten important classes in the program separately, and ran the ten versions of the program in parallel on separate machines, thus minimizing the slowdown. We could have eliminated all the invariant checks on run-time types, which tend to be the most expensive, on this program, since there was no polymorphism in the program. We did not do

```
for (replaced = 0; replaced < associativity; replaced++)
{
    // Bug - should have checked for 0 or 2
    if (status[replaced][curset] == 0)
        break;
}
```

Fig. 3: Sample code from multiprocessor simulator

so because DIDUCE did not support disabling these checks as easily at that time, and the overhead was not a significant limitation. We set up DIDUCE to use the initial part of each simulation run for training, and ignored the invariant violation messages it emitted in the training phase. Typically, the number of messages slowed to a small trickle after the first few minutes of execution.

We found almost all the violations detected after an hour of instrumented execution were interesting, with the exception of one annoying set of violations caused by a constantly increasing counter. The confidence value associated with these false violations was low, indicating to users that they may not be very important. Overall, DIDUCE discovered two bugs in the simulator that would otherwise be undetected and found the root causes of 3 other bugs. All the bugs were serious algorithmic errors, spread over 3 different classes which modeled different parts of the distributed and banked cache system. Besides identifying the 5 bugs found, the rest of the invariant violations pointed out about 10 rare corner cases. The programmer found these violations informative. He had to think hard to determine if these rare cases were, in fact, manifestations of bugs in the program. In some cases, he was surprised to learn that certain scenarios were so rare.

Fig. 3 illustrates one of the two bugs DIDUCE found that was not detected by any other means. This class in the simulator models a set-associative cache. The status of each cache line may carry the values 0, 1, or 2, to represent whether the cache line is empty, occupied or pending, respectively. A cache line is considered to be pending if it has been selected for replacement, but an invalidate from another processor for the same address reached this processor before the data could be fetched. The status variable is usually 0 or 1. The code fragment in Fig. 3 searches, among a set of candidates, for the first location whose status variable is 0. This is algorithmically incorrect, because the code should find instead the first location with status being either 0 or 2. The condition under which status is set to 2 is very rare, especially at this point in the code where a cache location is being selected to load a new memory block. The first instance of this occurs after slightly less than an hour of uninstrumented simulator run time. DIDUCE ran overnight on this program, identified an invariant violation for that line in the program, reporting that status = 2 violated the current hypothesis that its value should be either 0 or 1 at this point in the program. Upon reviewing this report, the programmer realized immediately that the code did not handle this rare scenario correctly. DIDUCE helped detect another otherwise-unknown bug with similar ease. This bug was related to a store being performed to a location in the cache whose state was invalid, which occurred even later in the program execution.

Apart from the two undetected errors, DIDUCE also helped the programmer find the root causes of three other bugs which were also detected by user-inserted assertion checks. In the first two of these cases, an invariant relaxation reported by DIDUCE correctly pinpointed the root cause to the exact line which contained the bug, though they were not particularly hard for the programmer to track down either. In the third case, the programmer could not understand the reason for the inconsistency reported by the assertion failure, which occurred after about an hour of uninstrumented execution time. He made several unsuccessful attempts to reason about what might have caused the assertion to fail. He tried to put in assertion checks and breakpoints in other parts of the code to try to catch the problem closer to its source. Each such debugging round took over an hour but they failed to yield any results. Finally, he gave up on the ad-hoc approach and used DIDUCE. Although DIDUCE did not identify the precise line where the bug was present, it identified a rare scenario through a series of six invariant violations which occurred just before the failure. Once the programmer found out about this scenario, he was able to identify the culprit immediately. Some of the invariant violations reported in this case by DIDUCE were simply about new code executed; therefore this debugging situation would also have been helped by a simpler tool which only reported incremental code coverage.

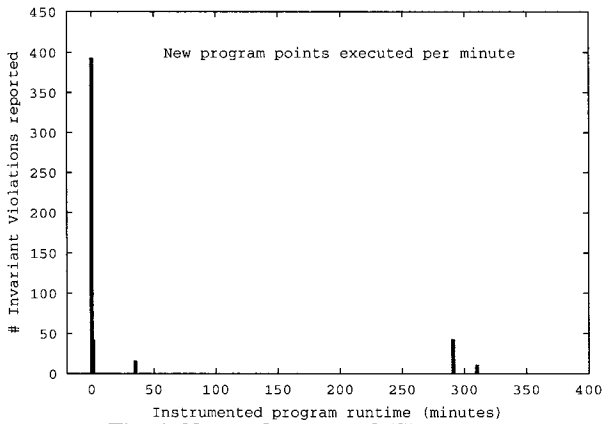


Fig. 4: New code executed (Simulator)

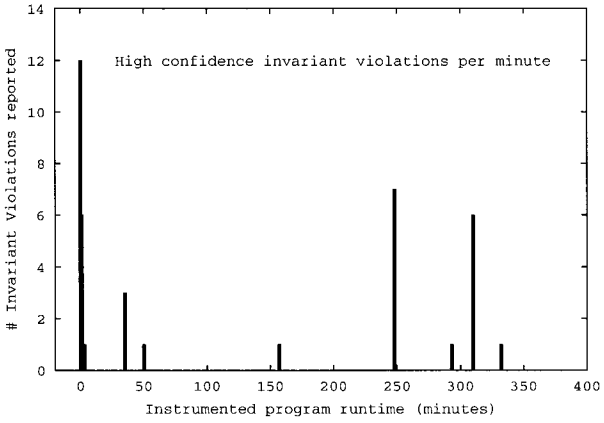


Fig. 5: High confidence invariant violations (Simulator)

In Fig. 4 and Fig. 5, we plot the number of invariant violation messages per minute of instrumented execution (for one of the 10 runs) in 2 categories - new code executed, and high confidence change invariant violations. These plots are especially interesting in this example because it is a long running program, and the early part of the program is used for training. Space restrictions prevent us from plotting such graphs for the rest of the examples we present. As can be seen from the graphs, many presumed invariants are violated in the beginning as DIDUCE tries to establish a model for the program.

In Fig. 4, the "new code" category tracks when execution reaches a program point for the first time, for the 531 program points instrumented in this run. Apart from these, there were a total of 710 violations reported, of which 40 were above a confidence change level of 100, and 83 above a change level of 10. In Fig. 5, the high confidence invariant violations refer only to the violations above the confidence change level of 100 (these include multiple violations for the same program point.) Since such invariant violations typically occur for expressions which have had a large number of evaluations and seen a small number of values, we usually find them interesting to analyze in most applications. Of course, lower confidence violations may also be useful to analyze. In practice, users typically sort the invariant violation messages and look at those with the highest confidence changes first, regardless of their absolute values. They also look at the last few newly instrumented program points.

The bug from Fig. 3 is represented by the last stub in Fig. 5 showing a high confidence change invariant violation about 332 minutes into execution (this violation had a confidence change of over 1 million, which was the highest during the whole run.) None of the other invariant violations plotted here was an actual error, although several of them gave the programmer interesting insights into the program run. As can be seen from these graphs, invariant violations tend to settle down quickly, and occur in clusters when new types of behavior are seen in the program.

Actually DIDUCE detected an invariant violation associated with yet another bug in the program (in a different class), even though we did not realize it at the time. The bug was independently discovered later. The DIDUCE warning went unnoticed because it happened fairly early in the program execution. The invariant violated in this case involved the variable representing the current simulation time. The hypothesized invariant at the time was that its new value must be larger than the old. DIDUCE warned of an invariant violation as it encountered the first case where the new value was actually smaller than the old, meaning that the simulation had moved backwards in time! Thus, while DIDUCE is successful in locating program errors, it is important that useful information not be buried in with the noise in the reporting.

5.2 Mailmanage

Mailmanage is an open-source email management utility to programmatically manipulate email mailboxes [14]. It makes extensive use of the JavaMail library, one of the extensions to the Java platform. While the program worked correctly on most mailboxes, it would crash on one particular mailbox after throwing a cryptic IO Exception. The crash apparently occurred in the JavaMail library while trying to fetch a message from a mailbox.

We used DIDUCE to instrument the Mailmanage program as well as the JavaMail library (for which we did not have the source code at the time), trained it on a few mailboxes for which the program worked correctly, and then ran it on the failing mailbox. DIDUCE printed out an invariant violation message, with high confidence, just before the program entered error-handling routines, which eventually threw the exception.

```
do {
    switch (buffer[index]) {
        case 'E':
            index += ENVELOPE.name.length;
            // other processing for case E
            break;
        // similar handling of other cases
    }
} while (buffer[index++] != ')' );
```

Fig. 6: Sample code from JavaMail library

We then obtained the source code for the JavaMail library, and looked at the program point identified by the invariant violation. The relevant code fragment, shown in Fig. 6, is a part of the library which parses the fetch response from the IMAP server. The entire response is placed in the buffer array; the fields of the response are parsed one by one, with the variable index always pointing to the beginning of a field within the response. The invariant violation reported by DIDUCE said that the variable buffer[index] contained a new value (10) when it was tested at the end of the while loop. The invariant established just prior to this violation accepts both the space character and the ")" character, which happened to denote the end of a field or the end of the response, respectively.

The bug in this case did not reside in the Mailmanage application and not even the JavaMail library, but the Solaris IMAP server. The root cause of the bug was that the IMAP server did not handle mailbox attachments created on a DOS file system properly. The response created by the server contained extra CR-LF characters, which caused an inconsistency with the length it reported for the RFC822 field in the message. This confused the JavaMail parser, which eventually threw an exception. This bug would have been very hard to find, without DIDUCE, for someone who had no familiarity with the inner workings of the JavaMail library.

This case study illustrates a few of DIDUCE strengths. First, it is able to use serendipitous invariants to identify the root cause of an error. In this case, while the use of a space character as a field delimiter is immaterial to the correctness of the program, it is exploited by DIDUCE to detect an anomaly in the input. Second, DIDUCE helps the user debug unfamiliar code, isolating the problem down to the component which actually contained the bug. DIDUCE correctly pinpointed the very line in the JavaMail library where the problem showed up. Third, DIDUCE helps find bugs in code that was not even instrumented by finding invariant violations at the interface between instrumented and uninstrumented domains. In this case, the root cause of the bug was in the Solaris IMAP server, which was implemented in a different language, running on a different machine. However, by adding instrumentation at the interface between the IMAP server and the JavaMail library, DIDUCE caught the error as soon as it propagated into the library. This effect can be used to detect errors in components which cannot be instrumented. It also lets us trade off instrumentation overhead against the accuracy of reporting exactly where a bug may lie.

5.3 The Java SSE Library

The JSSE (Java Secure Sockets Extension) v. 1.0.2 library is a mature piece of software that has been released and used for over one and a half years. The bug that led us to study this application was first noticed when a programmer tried to add a

proxy server to the library. She found that her changes triggered a previously unseen failure in apparently unrelated parts of the code. Perturbations made to the new code would exacerbate or alleviate the problem in an unpredictable fashion. She tried to debug this problem by working backwards from the point of failure through the rest of the library, which she was not familiar with. After 2 days of debugging, she had isolated the problem to a particular function, when we asked her to try using DIDUCE instead. She used DIDUCE to instrument the entire library and trained it on runs where the program had worked correctly. She then ran DIDUCE in checking mode on runs that failed. Tracing back from the point where the exception was thrown, she quickly found a high confidence invariant violation reported on the return value of a call to the SocketInputStream.read method. This part of the code is shown in Fig. 7. The return value of this method had always been equal to 74 on the training runs, since that was the value of the variable len for that version of the SSL protocol. The return value was different in the run that failed.

```
InputStream  s = x; // x is instance of SocketInputStream

// .. various SSL protocol processing

if (...) {
   int len = ... // expression for  length of header,
                 // always 74 at this program point
   byte[] hdr = new  byte[len];
   s.read (hdr);
}
```

Fig. 7: Excerpt from the SSL library

Focusing on this part of the code, it quickly became apparent that the writer of this code had fallen prey to a common Java pitfall. When the InputStream.read method is called with a byte array argument, it is not guaranteed to fill the array, even if the bytes are, or will eventually be, available. The call may return after it has filled in 1 or more bytes, as long as it returns the number of bytes actually filled in the array. The caller is expected to check the return value and keep re-executing the method call till the desired number of bytes have been read. This bug existed in the currently released version of the library, but was undiscovered because the array is in fact filled in completely most of the time. However, adding a proxy server to the library changed its timing and its behavior.

Once we knew about this problem, we modified DIDUCE instrumentation to include a simple static check for immediately discarded return values from calls to various flavors of InputStream.read with a byte array argument. We found over 80 such examples in the Java 2 Standard Edition and Enterprise Edition v1.3 libraries, most of which are likely to be errors.

This case study illustrates the importance of automatic invariant discovery. Having a fundamental misunderstanding of the java.io library interface, the original developer would not have placed an assertion on the return value. Had he thought that the check was needed, he would probably not have made the error in the first place.

5.4 Joeq

Joeq [11] is a large project which implements a Java Virtual Machine system with a just-in-time compiler. We instrumented the Joeq classes and ran the initial part of the virtual machine boot-up sequence in which the Java compiler system compiles itself. We ran DIDUCE in checking mode, without any training, and ignored the initial invariant violation messages.

```
JarInputStream in = ...;
Hashtable names = new Hashtable()...;
for (<each entry in the jar file> ) {
  JarEntry je = jin.getNextJarEntry();
  // process entry  ...
  names.put (je.getName());
}
assert (names.size() == jfile.size());
```

Fig. 8: Excerpt from joeq

Joeq failed an assertion while compiling a particular version of the Java Runtime Library. Joeq read each entry in the library JAR file, processed it, and entered the name of the entry into its own hash table. The assertion failure was caused by the fact that at the end of processing the file, the number of entries reported by the JAR file object did not match the number of entries it had in its own hash table. Whereas the assertion caught the fact that an error has occurred, DIDUCE pointed the programmer to the source of the problem precisely. The relevant code excerpt is shown in Fig. 8. As it turns out, the return value of the Hashtable.put method provides an indication of whether the object being inserted is already present in the hash table. It returns the existing object if the key matches an element in the hash table, and NULL otherwise. The programmer implicitly assumed that the entries in a JAR file were unique, and ignored the return value of the method. However, DIDUCE reported a warning when a duplicate was first encountered because the return value was not null for the first time. The failure had been caused by the fact that the library JAR file had duplicate entries for a particular file.

This example again corroborates the observation that there are usually plenty of clues that point to the source of the problem. Had the programmer not checked the number of entries in the hash table with an assertion, it would have been even harder to debug manually.

5.5 Summary Remarks

We applied DIDUCE to four very different applications, and it has been proven useful in every case we tried. It even discovered two errors in Simulator which would have gone detected otherwise. It helped find the root causes of many different kinds of errors.

- As illustrated by the Simulator case study, DIDUCE helps locate algorithmic errors that fail to handle corner cases correctly. As DIDUCE isolates the context in which an error occurs, the programmer can analyze the problematic scenario better and zoom in to the problem easily.
- As illustrated by the MailManage and Joeq case studies, DIDUCE helps user find errors in inputs, unfamiliar codes, and even uninstrumented components. The latter is achieved by noting violations of invariants governing the interface to uninstrumented domains.
- As illustrated by the SSL Library case study, DIDUCE helps identify errors due to a misunderstanding of interfaces between modules of a program. Programmers can quickly identify errors when presented with unexpected values observed at the interfaces.
- As illustrated by the Mailmanage and Joeq case studies, the automatic discovery of serendipitous invariants

enables DIDUCE to pick up many important clues that would otherwise be missed.

Another important side effect of the DIDUCE system is that users are informed about rare corner cases, which can be valuable in helping them understand their program better. Our experience suggests that our use of confidence levels is, for the most part, effective in singling out the noteworthy information. We have seen, however, one instance where the noise in the invariant violation reports prevented us from identifying an error.

6. RELATED WORK

The idea of detecting invariants automatically was inspired by the Daikon invariant detection system developed by Ernst et al [8]. They proposed dynamic invariant detection as a way to support program evolution, by helping programmers understand the code. The Daikon invariant detector runs an instrumented program and stores all the values taken by variables in the program run. An off-line analysis phase processes these values and checks for an extensive set of invariants at each program point, including properties of single variables, relationships between multiple variables, and other properties such as if an array is always sorted at a program point.

DIDUCE is designed to detect anomalies in programs to help programmers track down bugs and to find corner cases in programs. Our main contributions with DIDUCE are in scaling dynamic invariant detection to large programs, incorporating a systematic framework for dynamic invariant relaxation, and employing automatic, online checking of invariants, in addition to detecting them. Keeping the space of dynamic invariants small allows us to scale our implementation to large programs and carry out the invariant analysis online, enabling rapid feedback to the user.

Other dynamic bug detection techniques like Purify [10] and similar tools are widely used in commercial software development to detect unsafe programming practices like uninitialized memory reads, memory leaks and array bounds overruns. However, most of these errors are automatically avoided in a type-safe, garbage-collected language like Java. Eraser [18] is a dynamic analysis system which detects the set of locks protecting each variable in a multithreaded program and flags inconsistencies in the usage of locks. Various dynamic techniques have been proposed which use profiles of program runs to aid in program evolution[1][17].

Static bug detection methods attempt to analyze a program for possible bugs without running it. Static tools can verify that a program is correct for all inputs, whereas dynamic tools can only find errors triggered by input test cases. However, program verification is undecidable in general, and has only been applied successfully to small programs. Furthermore, static tools often require manual specification. Compaq ESC is a static checking tool that asks users to supply invariants at procedure interfaces and other key program points [5]. Experiences with the tool suggest that few programmers are willing to insert invariants into their code in real life. In contrast, DIDUCE is fully automatic; furthermore, in our experience, misconceptions are a common source of errors, i.e. the invariants supplied by programmer would have been incorrect even if he or she had tried.

Lackwit [16] allows users to assign finer distinctions to language types, by using the same declared type to represent multiple abstract data types. Type checking can then be performed on abstract data types, ensuring that the program uses the abstract types in a consistent way. Intrinsa's PREfix [2] is a tool which statically analyzes the program for undesirable properties like possible null pointers. PREfix uses path-sensitive analysis to explore multiple execution paths in a function, with the goal of finding paths along which undesirable properties can hold. Metal is a static analysis tool which allows users to write invariants about a program in a state-machine based language [6]. An enhanced compiler checks that these invariants hold along all possible execution paths. Metal has been successful in reporting several bugs in large pieces of code, such as the Linux kernel. The Vault system also allows programmers to describe resource usage rules and the compiler to check them [4].

Most of the bugs detected by the tools above are violations of simple API rules. Experiences with these tools suggest that even production software contains many such bugs, found mostly on program paths that have not been tested. To avoid overwhelming the users with false positive warnings, these tools tend to only report those that are very likely to be bugs. And even then, bugs that lie along infrequently executed code paths have lower priority, and may simply join the program's long list of outstanding bugs awaiting to be fixed. DIDUCE complements static approaches by finding subtle algorithmic bugs that occur on some inputs or after running a long time. These bugs are found along program paths that are executed often, but just never with a particular combination of inputs.

Nimmer and Ernst [15] propose a combination of static and dynamic approaches by feeding invariants hypothesized by Daikon to the static checker ESC-Java. If ESC-Java fails to verify a hypothesis, it suggests that the hypothesis may not hold true for some inputs. This approach can be used to find potential bugs if invariants are collected from correct runs and if ESC can then find the conditions under which the invariants are violated.

Our bug detection methodology with DIDUCE is similar in spirit to that proposed by Engler et al.[7] Their approach also tries to infer bugs by detecting inconsistencies from commonly observed behavior. While they follow a purely static approach by looking at structures in the code, we detect anomalies in dynamic program runs. Furthermore, their tool is still limited by the types of errors which are coded in the analysis.

7. CONCLUSIONS

This paper proposed the idea of finding program anomalies through an on-line dynamic program invariant detection and checking engine. Our experimentation with four real-life applications suggests that DIDUCE is effective in detecting hidden errors and finding the root causes of complex programming errors. It can find bugs that result from algorithmic errors in handling corner cases, errors in inputs, and developers' misconceptions of the APIs. It helps programmers locate bugs in unfamiliar code and, sometimes even in codes that have not been instrumented. Furthermore, no up-front investment is required; users start using DIDUCE only when they are confronted with a bug, or the possibility of one. While we used only the simple, default invariants in our experiments, users can tailor DIDUCE to check for more complex invariants to suit the specific application.

In the future, we wish to lower the run-time overhead of DIDUCE so that it can be used to monitor applications in use. Coupling this approach with an error recovery mechanism will make software more resilient to failures. We also intend to explore the use of anomaly detection to aid in the software evolution process. An automatic tool that can identify the subtle differences in the behaviors of programs before and after modification would be invaluable.

8. ACKNOWLEDGMENTS

We thank Shailender Chaudhry, Jaya Hangal and John Whaley for providing test cases and giving us feedback about using DIDUCE. This research is supported in part by NSF award #0086160.

9. REFERENCES

[1] T. Ball. The Concept of Dynamic Analysis. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'99)*, pp. 216–234, September 1999.

[2] W. R. Bush, J. D. Pincus and D. J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software Practice and Experience,* Vol. 30, No. 7, pp. 775–802, 2000.

[3] ByteCode Engineering Library. http://www.sourceforge.net/projects/bcel

[4] R. DeLine and M. Fahndrich. Enforcing High Level Protocols in Low Level Software. *Proceedings of PLDI, June* 2001.

[5] D. L. Detlefs, R. M. Leino, G. Nelson, J. B. Saxe. Extended Static Checking. SRC Research Reports SRC–159, Compaq SRC, December 1998.

[6] D. Engler, B. Chelf, A. Chou and S. Hallem. Checking System Rules Using System–Specific, Programmer–Written Compiler Extensions. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation*, October 2000.

[7] D. Engler, D. Chen, A. Chou, S. Hallem, B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in System Code, in *Proceedings of Symposium on Operating Systems Principles*, October 2001.

[8] M.D. Ernst, J. Cockrell, W.G. Griswold and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *IEEE Transactions on Software Engineering*, Vol. 27, No. 2, February 2001.

[9] European Space Agency. Arianne–5 flight 501 Inquiry Board Report. http://ravel.esrin.esa.it/docs/esa–x–1819eng.pdf

[10] R. Hastings and R. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter Usenix Conf.* pp. 125–136, January 1992.

[11] Joeq. http://www.sourceforge.net/projects/joeq

[12] N. Leveson and C.S.Turner. An Investigation of the Therac–25 Accidents *IEEE Computer*, Vol. 25, No. 7, July 1993.

[13] Lindholm, T and Yellin, F. The Java Virtual Machine Specification, 2nd Edition, Addison Wesley, April 2000.

[14] Mailmanage: http://www.sourceforge.net/projects/mailmanage

[15] J. Nimmer and M. D. Ernst. Static Verification of Dynamically Detected Program Invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV–01, The First Workshop on Runtime Verification*, July 2001.

[16] R. O'Callahan and D. Jackson, Lackwit: A program Understanding Tool Based on Type Inference. In *Proceedings of the 19th International Conference on Software Engineering,* pp. 338–348, May 1997.

[17] T. Reps, T. Ball, M. Das, and J.Larus. The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In *Proceedings of the Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'97),* pp. 432–449, September 1997.

[18] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T.. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* Vol. 15, No. 4, pp. 391–411, November 1997

[19] M.Tremblay, J.Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC Architecture: A Synthesis of Parallelism and Scalability. *IEEE Micro,* Vol. 20, No. 6, November/December 2000.