Generic Invariant-Based Static Analysis Tool for Detection of Runtime Errors in Java Programs

Sotiris Skevoulis
Department of Computer Science
School of Computer Science
and Information Systems
Pace University
sskevoulis@pace.edu

Xiaoping Jia
School of Computer Science,
Telecommunication and
Information Systems
DePaul University
xjia@cti.depaul.edu

Abstract

This paper presents an invariant-based generic tool to statically analyze Java programs in order to detect potential errors (bugs). We briefly discuss the supporting theoretical framework and highlight the results of the tool in statically analyzing Java code. The tool can automatically detect potential bugs such as illegal dereferences and array bounds and report them before the program is executed. For a Java class, invariants related to the category of error under examination are automatically generated and used to assess the validity of variable usage in the implementation of this class. The tool provides a practical and extensible generic mechanism for error detection to help industry practitioners who work with an object oriented language such as Java. The presented mechanism is capable of addressing error detection for a variety of error categories that can not be caught by flow-based static analysis tools.

1 Introduction

Error detection in programs has always been one of the most active areas of research in computer science. Hoare triples[9], Dijkstra's weakest precondition [2] and recently complete verification environments such the Prototype Verification System (PVS) [16] are just some exemplary examples. Despite that, software industry still faces formal methods with a certain degree of skepticism and programs still are written and delivered with a large number of problems which cause embarrassing, costly and life-threatening failures [15, 11, 4].

Programming language developers have also tried to address the problem of error detection in the source code. Sophisticated programming notations with smart compilers attempt to notify the user in a safe manner when errors occur. Examples of such mechanisms include *strong typing* and *runtime checking*. Java[5] is an example of such programming language, which is strongly typed with extensive and sophisticated runtime checking mechanism in the form of runtime exceptions.

However, even the most sophisticated compilers have a limited ability to detect potential code errors. This is due to certain time, resources and performance constraints that limit the amount of checking that compilers can perform. Still errors such as illegal dereference, violation of array bounds etc, cause interruption of program execution and cannot be caught by any compiler. Most of these kinds of errors are usually caught during runtime. But catching an error at runtime offers little on no advantage at all.

Traditionally static analysis has been based on control flow based techniques. Such techniques do provide some limited capabilities in catching errors but since they do not use any reasoning in the processing of source code, often enough generate false positives, or miss errors altogether. On the other hand, formal verification techniques usually come with unacceptable costs. Our approach bridges two areas: formal methods and static analysis under one unified and extensible framework.

We approach the problem of potential error detection, using a lightweight theorem proving formal methods approach, with more pragmatic goals. We drop the guarantee of correctness in order to deliver a partial solution that will detect the majority of the potential errors we search for. We apply static analysis and verification techniques, to automatically analyze Java programs and detect potential bugs that cannot be detected by compilers and other data flow-based analysis techniques. Our approach is based on the use of the logical concept of a *class invariant*, which is a set of invariably true statements concerning the variables of a class and *weakest precondition* [6]. An automated lightweight theorem prover provides the reasoning mecha-

nism in our framework.

Java language has been selected as the target language in our research, because it is quickly becoming one of the most widely used programming languages in software industry and it is more manageable than C and C++, since the semantics of its control flow is simpler without the goto statement and no pointer arithmetic is allowed. This fact makes alias tracing feasible. Our framework consists of the following components:

- *Generic analysis* It provides the analysis framework for a variety of error categories. It is extensible and customizable
- *Specialized analysis.* It specializes the behavior of some of the steps defined in the generic mechanism. Currently, specialized algorithms are provided for null pointer and array bounds analysis

Our analysis is targeting a variety of errors that include illegal dereference, array bounds violations, illegal downcasting and string index out of bounds. Initializations of pointers, proper handling of them, and proper indexing of arrays are very important issues in order to avoid unexpected and embarrassing program crashes. The common steps in the detection process for these errors has been captured in our generic analysis framework, while certain key behaviors that differ from error to error have been captured into specific customized algorithms that take into account the idiosyncratic behavior of each construct i.e. pointer and array. The key advantages and characteristics of our approach could be summarized as follows:

Effectiveness. Our approach is capable of detecting errors which can not be detected by flow based approaches:

- Detection of errors only within the *feasible* paths of program execution. If an error occurs within an unexecutable path, it is not reported
- Identification of errors that are parts of conditions. Variable dereferences and array accesses could be found in conditions and such errors are detected
- Guarded use of variables which may cause errors are not reported. If an illegal dereference or an array access is guarded by the appropriate condition, false positive responses are not generated

Extensibility. The design of our solution guarantees the extensibility of the approach. A common core of actions in order to determine the appropriate kind of invariant for a variety of errors has been captured. Extensive use of *strategy* and *template* design patterns, enables many specialized components to support the invariant determination and check violation algorithms. Because of the flexibility gained by the applications of design patterns, customizable components and/or algorithms can be easily added to the framework to check for more error categories

Generic detection mechanism. The approach is both specific enough to detect errors deterministically and generic enough to provide an efficient and effective framework for a variety of error categories

Complete automation. No effort is needed from the software practitioner. The entire process of analyzing the source code and reporting potential violations is fully automatic.

Complete Java coverage. Neither syntax nor semantic restrictions has been imposed on the original language.

The detection mechanism of our approach has certain limitations, mainly related to the intractability of theorem proving process. We cannot guarantee absolute success in finding *all* the code anomalies even for the restricted types of analysis discussed earlier. Our goal is not to find *all* errors rather to find the majority of them in a fully automatic and transparent to the developer way.

A fully functional prototype tool that implements the algorithms and demonstrates the feasibility and effectiveness of our approach has been developed. We used the tool to run an extensive number of test cases for both types of specialized analysis (null pointer and array bounds) and the results were very encouraging. Our approach did effectively detect the majority of program anomalies in the test cases.

2 Motivating Examples

Java programs heavily depend on the use of reference variables. Such variables need always to point to valid memory locations otherwise any attempt to dereference them will lead to null pointer exception. There are many reasons that may cause a pointer to point to null. For example they may have never been initialized to anything, or may become accidentally null in the course of program's execution etc. Let us consider the following example:

```
1.public class AClass {
2.
3.  private String s1 = new String();
4.  private String s2 = new String();
5.  private String s3;
6.  static String str=new String("A String");
7.  private int x  = 0, y, z;
8.
9.  public AClass(int i) {
10.   y = i;
11. }
12.
13. public String aMethod(int k) {
14.   while (x > 0) {
15.     if (k == x) {
16.       s3 = new String("A string");
17.       break;
18.     }
19.     x--;
20.   }
21.   System.out.println("The string is: " +
                            s3.toString());
22.   return s3;
23. }
24.
25. public void bMethod() {
26.   if (x > y){
27.     System.out.println(s1.toString());
28.     if (y < x) {
29.       s2 = null;
30.       System.out.println("The size of s2 :" +
                                    s2.length());
31.     }
32.   }
33. }
34.
35. public void cMethod() {
36.   int z = 10;
37.   if (s3.length() > z) {
38.      s3 = s2;
39.   }
40.   System.out.println(str.toString());
41. }
42.}
```

The above Java class contains a few strings some of which are initialized and some others are not. Notice that in method `aMethod` string `s3` is initialized only under some condition. If this condition, for some reason, is never met, then the subsequent dereference will cause a null pointer exception. In method `bMethod` the string `s2` becomes null just before the dereference, in the second `if` statement. But since, the condition $y < x$ will never be satisfied, (it is an unexecutable path), in fact there should be no potential violation there. Finally in method `cMethod` the variable `s3` is dereferenced in the `if` statement condition. String `s3` has never been properly initialized and will cause again a runtime exception. The above situations is an indicative sample of problems which can very easily arise in programming with Java and have not been caused by any major flaws in design. Nevertheless, they will cause abnormal program interruption if not caught early on in the development process.

Let us now turn our attention to the array usage in programs. Consider the well known *Bubble Sort* algorithm shown below.

```
1.public class BubbleSort {
2.  String name = new String("Bubble Sort");
3.
4.
5.  public void sort(int[] arr) {
6.    System.out.println("Sorting array with " +
                                    name.toString());
7.    int i = arr.length;
8.    while (i > 0) {
9.      int j = 0;
10.     while (j < i) {
11.       if(arr[j] > arr[j+1]) {
12.         swap(arr, j, j+1);
13.       }
14.       j = j + 1;
15.     }
16.     i = i - 1;
17.   }
18. }
19.
20. public String getAlgorithmName() {
21.   return name;
22. }
23.
24. public void swap (int[] a, int i, int j) {
25.   int temp = a[i];
26.   a[i] = a[j];
27.   a[j] = temp;
28. }
29.}
```

Notice that the programmer does not initialize the `while` loop control variable `i` in line 7 properly. Programmer failed to set the index i to the appropriate initial value of `arr.length - 1`. It is a kind of mistake, which can occur very often in every day software development. Of course at runtime, it will lead to an array index out of bounds exception and will cause abnormal termination of program's execution.

3 The analysis mechanism

The generic analysis is carried out mainly by two algorithms: *DetermineInvariant* and *CheckViolation*. They analyze a Java class in order, first, to determine an invariant

and subsequently use it to check for any potential violation in the implementation of this class. They extract the specifications related to the kind of error under detection directly from the source code.

Determining a class invariant regarding specific properties involves the construction of a candidate invariant and subsequent checking to determine whether or not satisfies the requirements to be an invariant. In order for a condition to be part of the invariant it must be true for all non-transient states of the instances of a class. The fundamental design assumption for the invariant determination algorithm is that it has to be generic enough to allow generation of invariants related to different categories of error. The input to the algorithm is a Java class and the output is an invariant property (predicate), about this Java class. The invariant determination process is made up of the following phases:

Construction. Given a Java class, its appropriate fields (reference types, array variables, etc) are read in, both static and non-static, and a candidate invariant is formed.

Break down. The formed invariant is broken down into two predicates: one that expresses a condition regarding static variables *only* and one about all the class level variables. Invariant is broken down into two sets in order to examine separately the properties that do not depend on the instantiation of the Java class from the ones that do.

Mutation. The invariant is mutated in a number of different forms. The goal is to provide an array of weaker predicates and store them in order of strength starting from the strongest one (the predicate itself unchanged) going to weaker ones until the predicate cannot be mutated anymore and its weakest mutated form is the predicate `true`.

Verification. Each of the mutated predicates is examined separately and if it fulfills the requirements of an invariant, is added to the invariant of the class. An invariant property has to be ensured by construction blocks (initialization blocks and constructors) and preserved by each non-private method.

For example, for a reference variable `var`, the predicate that will be examined could be $var! = null$. If it does not qualify as an invariant, the mutation process returns just *true*. Let us assume, we have a variable of array type `myArr` with size `aSize` where `aSize` is some integer or a variable of type integer and we try to detect any array index out of bounds violation. A candidate invariant could be the predicate: $myArr.length \geq size$. If this predicate cannot be proved to be an invariant, then we can always lower the value of the size. This process is *deterministic* because there is always a lower bound which is the number `0`.

Specialized algorithms for construction and mutation proacesses have been developed that take into account the different idiosyncratic properties of each category. These include the construction of predicates i.e. different predicates are formed for null pointer and array bounds analysis.

The check violation algorithm checks the usage of each relevant variable to detect any potential violations. In order to do that we provide an algorithm that scans though the implementation of the class, locates all the relevant variables and forms the appropriate verification conditions. This is shown in a general form below:

```
void aMethod(...) {
   {Invariant}          the pre-condition
   // ...               other statements
   {v! = null}          the assertion we attempt to prove
   v.m(...)             the dereference
}
```

3.1 The Specialized Analysis

Null pointer information is essential to knowing the state of the pointer at an arbitrary point in a program for conducting other analyses (array index out of bounds, class cast exception). Software engineering techniques such as program slicing need null pointer analysis as well. In null pointer analysis, predicates which are parts of the invariant will have to be in the form:

$$var! = null$$

where var is a reference variable of the Java class. A specialized algorithm will attempt to construct an invariant concerning the class level reference variables in the class. The rest of the specialized algorithms will eventually determine an invariant regarding the instance variables of the class.

The check violation process needs to establish what variables need to be checked. In other words, we need a collection of the variables that are dereferenced at any given point in the source code. Another specialized algorithm searches through each single statement and identifies which variables need to be checked. As soon as the verification conditions are generated, they are forwarded to the theorem prover. If the verification condition gets discharged,

then the usage of the particular variable is safe. If it is not, then we *may* have found a potential violation.

The array index checking process involves the creation of the proper candidate invariant. In this case we are interested in a predicate regarding the size of the array that can be used as invariant. For array *arr* with length *arr.length* and initial size *size*, the algorithms attempt to formulate a predicate in the form:

$$arr.length \geq size$$

The process of determining *size* requires a specialized algorithm that will symbolically execute the init blocks and the constructors to determine an initial size for the array. This initial size will be used to form the candidate invariant for this array, which will be verified against init blocks, constructors and public methods in the given class. An array can be initialized either via an *ArrayInitializer* expression, which specifies the size of the array or via aliasing a previously initialized array. The semantics of array initialization are captured in our approach by three rules. Using these rules we cannot only determine an initial size but also we can monitor the various changes of the size of a given array that occur throughout the entire code. Let us assume that *arr* and *brr* are two arrays and *size* is an integer variable, then:

- $arr = \{e_1, e_2, \ldots, e_n\}$ is interpreted as $arr.length = n$
- $arr = brr$ is interpreted as $arr.length = brr.length$
- $arr = newType[size]$ is interpreted as $arr.length = size$

During the checking violation process, every time we examine a Java statement, we need to distinguish the variables of interest from other variables in the statement. A specialized algorithm selects those variables that we are checking and returns them to the CheckViolation algorithm[10, 17]. The variable selection will collect all the array access variables and present them in the violation detection mechanism to check their validity.

4 Error detection

In the examples presented in section two, the analysis process will start with the algorithm that determines the class invariant will accept as input the Java class `AClass`. Let us assume that we are about to analyze the class for potential illegal dereferences. The first step in the algorithm involves the construction of two sets of predicates representing the candidate static and instance invariants involving the reference variables namely: `str`, `s1`, `s2`, `s3`.

The specialized algorithm that constructs the candidate invariants will process the class level variables, it will select the ones of reference type and distinguish any static ones for the static candidate invariant[10, 17]. The output will be: a) for the candidate static invariant $\{str! = null\}$ and b) for the candidate instance invariant $\{str! = null, s1! = null, s2! = null, s3! = null\}$. The set representation for each predicate means the logical conjunction of the predicates-members of the sets. Each one of the above sets will be broken down to a number of combinations, forming several different predicates, which will be tried to verify whether they fulfill the requirements to be part of the class invariant. The task of checking that is accomplished by two similar algorithms: *IsStaticInvariant* and *IsInvariant*. The first one will verify that $str! = null$ is indeed an invariant property regarding the static variables of the class. The output of the *IsInstanceInvariant* will be $\{str! = null, s1! = null, s2! = null\}$.

Notice that even though $s3$ is initialized in one of the possible paths of execution of method `aMethod` (line 16.), this fact alone is not enough to make this predicate part of the invariant of the class. Also, notice that the reasoning power of our approach successfully recognizes that the nullification of variable `s2` in method `bMethod` (line 29.) *does not* affect its proper initialization, because it becomes null in an unexecutable path. The conditions $x > y$ and $y < x$ are contradictory. This is one of the results that flow based approaches cannot achieve, due to the lack of reasoning power.

The invariant that will result from the invariant determination process will be : $\{str! = null, s1! = null, s2! = null\}$. Having this predicate representing the invariant property, the checking violation process can begin. In each line of class's implementation, the variables of interest (i.e. reference variables) will be selected and the appropriate verification conditions will be formed. Then, given the class invariant, a weakest precondition calculation process will start from that point backwards to establish that the invariant property is enough to support the truth of the verification condition at that line of the code. For example, the safety of the variable dereference for `s3` in line `21.`, will require the proof of the following verification condition: $s3! = null$. Given as a premise the invariant of class and the statements preceding the one in line 21, this condition cannot be proved by the theorem prover. Thus there is a good possibility that this particular dereference might lead to a runtime exception. The output of the tool is shown

below:

```
Potential Null Pointer Violations
The static invariant for class:
              AClass is : str != null

The invariant for the same class is:
   str != null && s1 != null && s2 != null

Violation for variable:  s3  at:
Line: 21  System.out.println("The string
                     is: " + s3.toString());
Line: 37  s3.length() > z
```

In the second example the bubble sort algorithm fails to properly initialize one of the two loop control variables (i.e. `i`) in line 7. This will result in an array index out of bounds few lines later (line 11) when the index `j+1` will go beyond the array index limit. The verification condition for the array access expression in line 11 will be $j + 1 < arr.length$. In one of the paths of execution, the conditions in order to reach the array access in line 11 form the following predicate: $i > 0 \Rightarrow j < i \Rightarrow j + 1 < arr.length$. The appropriate weakest precondition calculation with the textual substitution [6] will lead to the following condition: $0 < arr.length \Rightarrow 1 < arr.length$ which clearly is false and a potential violation has just been found.

```
Potential Array Bounds Violations
The static invariant for class:
               BubbleSort is : true
The invariant for the same class is:
               arr.length >= 10
Violation for variable:  arr  at:
Line: 11  if (arr[j] > arr[j + 1])
```

For any kind of potential error they we are trying to detect (null pointer, array bounds, etc), our approach is effective in reducing the number of spurious errors (false positives). The effectiveness of the detection mechanism is influenced by the capability of the theorem prover. The theorem prover may fail to prove certain candidate theorems which actually hold, due a number of reasons related to the limitations of the automatic proving process. This will result in either an invariant failed to be proved during the generation of invariants or an error being signaled during error detection process, which it is not actual error. In essence, we have moved as much as possible the burden from the theorem proving by extensively analyzing the source code and transform it into a much simpler representation[17]. Overall, we obtained less that 5% false positive results on the extensive set of test cases that we run our prototype.

Consider the lines 26-29 in `AClass` example in section two. It shows a scenario with a high potential to lead to false positive results. In method `bmethod` the string `s2` becomes null just before the dereference, in the second `if` statement. But since, the condition $y < x$ will never be satisfied, (it is an unexecutable path), the tool will not produce a warning message. In other words, the tool is smart enough to identify and ignore potential violation in unexecutable paths.

5 Experiments

We assess the capabilities of our approach by exposing it to a number of small programs that provide clear evidence that it substantially improves software practitioner productivity and can serve as a vehicle for further research in the area. Our first goal was to demonstrate the feasibility of the approach. Small-scale projects have been tried and the results have been evaluated. Our research effort is experimental in nature. Its success can be judged through experiments rather than theoretical proofs and analyses. The experimental prototype allows us to gain insight, discover obstacles and limitations of our techniques. We use the feedback given by the tool to refine and enhance our approach. The assessment phase is based on the following steps:

- Design a method to gain data and eliminate any bias.
- Measure the frequency of different kinds of errors in the programs.
- Gain evidence in a controlled environment that our technique finds the majority of those errors.

We have developed a classification scheme for the different causes of errors in both kinds of specialized analysis discussed in this paper. For each kind of anomaly we analyzed and enumerated the causes and the scenarios under which a violation will occur in the program. Prior to any testing all test cases were successfully compiled to ensure syntactic correctness of the code.

For each of the classified cases we performed complete set of permutations under different scenarios to measure the effectiveness of the approach. Our sample consists of about 200 Java classes from projects developed both in classroom environment and in the inustry setting. Test cases for a predefined confidence level were generated, using templates and applied to each mutated version. The generation of new mutated versions of the program was repeated, with

Category	Scenario	# Faults	# Found	# Missed	# False Posit.
Illegal dereference	Initialization	40	36	4	1
	Random Nullification	50	45	5	2
Array Bounds	Initialization	20	20	0	0
	Local scope	50	48	2	1
	Class scope	40	28	12	3

Figure 1. Metrics from test cases

different faults under different context's until enough samples were collected to evaluate the prototype performance in a meaningful way. A detailed chart which presents the performance of the tool under the different cases and scenarios is shown in Fig. 1. In most of the cases that our tool failed to indicate an error, the problem was isolated in the intractability of the proving process. Currently the experiments are based on *intra* class, *intra* method scenarios.

The illegal dereference set of test cases has been divided into two major categories. Several subcategories have also been included. A brief description of the major categories follows:

- Lack of initialization.
  Initial tests conducted were to check for null pointer violations in programs where variables were uninitialized. Test cases included both static and instance reference variables within any kind of executable Java code (i.e. initialization blocks, constructors, methods).
- Variable initialized but is nullified at arbitrary point
  There are quite a few ways that an initialized variable may get to point to null. Possibly, it is accidentally aliased by another null variable via parameter passing, or the programmer assigns it directly.

The array bounds testing scheme categorizes the potential errors according to the informal semantics of the error and the distribution of array access usage. We conducted a study on the different ways that arrays are used and we categorized their usage in a number of categories . The vast majority (more than 90%) of the arrays created and used, fall into the categories we present in the following sections. We have identified the following scenarios:

- Array initialization scenario.
  Errors in this category can occur in a variety of ways such as array size is initialized with negative integer (possibly 0), or it is initialized with integer variable the initialization sequence of which leads to negative or 0 value.
- Array is created and accessed within local scope scenario.
  Our approach detects almost all array bounds bugs within local scope. In this case, no invariant determination process needed and the detection mechanism can automatically point out any error.
- Array declared as class or instance variable scenario. In this case, an invariant regarding the size of the array is determined. In this scenario we also need to distinguish a three different cases.
  - Array initialized with integer constant and the size remains constant.
  - Array initialized with integer constant but the size changes in various constructors.
  - Array is initialized with integer variable.

The reduction of the generation of false positives has been set as one of the most important goals of the approach. A tool that generates too many false positives will be discarded by software practitioners. In order to demonstrate tool's effectiveness in reducing the generation of false positives we tested it in a number of cases with increased likelihood of generating a false positive:

- Guarded use of variables.
  Both reference variables and array access expressions which ordinarily would have forced the prototype to signal an error, have been placed within conditions which guard the dereference or the array access. In all the experiments the prototype did not indicate a false positive.
- Unexecutable paths.
  Null variables and/or array indices out of bounds have been placed in unexecutable paths. The prototype tool was intelligent enough to ignore these as potential errors.

In summary, the early experiments are encouraging. The prototype is still under development and more extensive case studies are planned.

6 Related work

Over the past twenty years there have been approaches that intend to verify fully functional correctness of programs. For that purpose, programming languages specifically invented for those approaches or subsets of widely used languages. Examples in this category include AFFIRM [14], Stanford Pascal Verifier [7].

Flow control based techniques have been used successfully as part of modern compilers such as Java, to provide early detection of a various anomalies and minimize the number of inconsistencies in the programs. Unfortunately these techniques lack in reasoning capabilities, which are necessary in order to a variety of potential bugs. These kinds of problems are beyond the capabilities of compilers and subsequently are transferred to the runtime environment and manifest themselves as abnormal program terminations.

Proof based static analysis has been carried out recently, by methods such as abstract interpretation [1], and Set Based Analysis [1, 8]. In these methods, the invariants are established about a set of values that a variable or expression is allowed to assume for assignment. A method to explain and present those invariants is reported by Flanagan [3], and is based on Set-Based Analysis and a mark-up notation for easy visualization of the analyzed program. More recently, research work at DEC laboratories has resulted in *Extended Static Checking* (ESC) [13] and checking object invariants [12]. This research effort attempts to use formal methods to identify particular kinds of bugs in a programming language and provide also some kind of feedback to the programmer about those potential bugs. ESC translates programs into an abstract form based on Dijkstra's guarded command. Its logical framework is untyped first order predicate calculus and the lack of type information for the proofs is covered by the background predicate [12] which includes type axioms. An unsuccessful proof returns information about the reason of its failure.

Our approach most closely resembles the work at DEC. However, there are major differences between ESC and our approach. ESC requires programmers to provide specifications as annotations in order to carry out the analysis while we do not require the presence of any kind of specifications. Also, the abstract form representation, although simple and elegant, it increases the burden of the theorem prover. We incorporate techniques such as symbolic execution and simplification of algebraic expression before and during the proving process, in order to automate it.

7 Future work

The work presented in this paper lays out the foundation of a comprehensive analysis of Java classes. Our work will evolve in two main directions in order to provide:

- Stronger analysis techniques which will cover more complicated cases within the already defined categories of potential bugs and extension to inter class - inter method analysis.
- Broader coverage of kinds of errors that can be automatically checked. These include but not limited to illegal downcasting, string index out of bounds

Each one of the above new challenges will create more complicated candidate theorem to be proved. Our research effort will continue to develop new heuristics, decision procedures and tactics that will extend the capabilities and empower our theorem prover.

8 Conclusion

Usually, checking for errors is not one of the first priorities of a software developer. Very often, highest priority is given to meeting unrealistic deadlines by working around the clock rather than making sure that the developed code is free of bugs. As a result, code is released into testing before adequate checking is done. Sometimes errors even escape detection during testing and products are released to customers with a number of bugs hidden in the code. Our approach provides a solution to this problem by focusing on the detection of certain kinds of bugs. The approach and algorithms described in this paper are dealing with the concept of class invariant and based on that, check the implementation of the class. The implication is that for an instance created by any public constructor of the class, any public (thread-safe) method can be invoked in any order. In our work, we address some of the obstacles that prevent formal methods from being widely used in software industry. We make the use of formal methods and theorem proving completely transparent to the software practitioner. By sacrificing guarantee of absolute correctness and lack of errors, we provide a mechanism to detect an arguably significant number of common errors, that are part of the daily debugging routine of every software practitioner. The prototype tool under development provides the means for measuring the effectiveness and capability of our approach. Early experiments indicate that:

- Our approach does not introduce any radical changes in the way software practitioners develop their code
- The analysis is fully automatic and removes the burden of proof from the software practitioners

We have focused on certain types of anomalies in the source code that *can* be detected automatically. The key feature which makes our approach practical is its complete automation which entails practicality.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis off programs by construction or approximation of fixpoints. In *Proc. ACM SIGPLAN Conference on Principles of Programming Languages*, pages 238–252, 1977.

[2] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of program. *Communications of ACM*, 18(8):453–458, 1975.

[3] C. Flanagan et al. Static debugging: Browsing the web of program invariants. In *Proc. PLDI96*, 1996.

[4] W. Gibbs. Software's cronic crisis. *Scientific American*, pages 86–95, 1994.

[5] J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.

[6] D. Gries. *The science of Programming*. Springer-Verlag, 1981.

[7] S. V. Group. Stanford pascal verifier user manual. Technical report, Stanford University Computer Science Department, 1979. STAN-CS-79-731.

[8] N. Heitze. Set based analysis of ML programs. In *Proc. ACM SIGPLAN Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[9] A. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12(10):576–580, 1969.

[10] X. Jia and S. Skevoulis. A generic approach of static analysis for detecting runtime errors in java programs. In *Proc. 23th Annual Int'l Computer Software and Applications Conf.*, pages 74–79, Phoenix, Arizona, USA, Oct. 1999.

[11] D. Kemp and G. Goodfellow. At&t crash, 15 jan. 1990. the official report. Technical report, ACM SIGSOFT, 1990. Software Engineering Notes.

[12] K. R. M. Leino. Ecstatic: An object-oriented programming language with axiomatic semantics. In *Proc. FOOL4*, 1997. Fourth International Workshop on Foundations of Object-Oriented Languages.

[13] K. R. M. Leino and R. Stata. Checking object invariants. Technical report, Digital Equipment Corporation Research Center, 1997. Palo Alto, CA.

[14] D. Musser. Abstract data type specifications in the AFFIRM system. In *Proc. The Specification of Reliable Software*, pages 47–57, 1979.

[15] N.G.Leveson and C. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 7(26):18–41, 1993.

[16] N. S. S. Owre, J. Rushby. PVS: A prototype verification system. *Lecture Notes in Artificial Intelligence*, 607:748–752, 1992.

[17] S.Skevoulis. A light-weight approach to applying formal methods in software development. Technical report, DePaul University, 1999. PhD dissertation. Available as Technical Report from DePaul University, `http://www.cs.depaul.edu`.