Detecting Null Pointer Violations in Java Programs

Xiaoping Jia, Sushant Sawant
Jiangyu Zhou, Sotiris Skevoulis
Division of Software Engineering
School of Computer Science, Telecommunication,
and Information Systems
DePaul University
Chicago, Illinois, U.S.A.
E-mail: `jia@cs.depaul.edu`, `ssawant@us.oracle.com`,
`jzhou@shrike.depaul.edu`, `sskevoul@cs.depaul.edu`

Abstract

The use of formal methods has been growing steadily and there have been a number of successful applications of formal methods in a range of application areas. It seems agreed that quality should be assured by applying testing, analysis and formal methods to rigorously defined precode artifacts. The detection of null pointer violation errors is definitely such a goal. This way of applying formal methods has a great potential to increase our confidence in the software. Our goal is to provide a practical mechanism to assist the application of formal methods in the early detection of null pointer violation errors in programs. Our solution is theorem proving based and is focused on the identification of the possible places in which a theorem prover could assist in the detection of null pointer violation errors and the formulation of the necessary proof obligations.

1. Introduction

The strong type system that most programming languages use cannot prevent tedious bugs from occurring. Such bugs include dereferencing a null pointer or an index falling out of bounds etc. These kinds of problems are beyond the capabilities of compilers and subsequently are transferred to the run-time environment where some languages offer mechanisms to deal with these problems (exception handling mechanisms). In general, the term static analysis is used to describe a spectrum of tasks that vary from enabling compilers to generate better code [8, 10] to the discovery of code anomalies or deficiencies [9] and the reduction of inconsistencies within the program.

Over the past twenty years there have been approaches [1] that use assertions that have to be true at some point or points during program execution. Such approaches can be found in LCLint [2] that uses annotations to perform error checking. *Anna* [7] uses mathematical description of certain program properties inserted in ADA code as formal comments. Ongoing research work at DEC laboratories has resulted in Extended Static Checking (ESC) [4, 5] and checking object invariants [6]. They attempt 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.

In this paper we present a theorem proving based static program analysis technique that is capable of detecting illegal dereferences in Java, Our technique is completely automatic and does not require the programmer to provide specifications, although it could be helpful. The theorem prover helps in evaluating the implications of conditions defined in the program.

Our approach is based on the concept of weakest precondition as defined by Gries [3]. The technique attempts to formulate certain global program properties from the class under examination and evaluates them with a theorem prover. These properties are classified as class invariants. The class invariants are combined with the precondition for a variable under examination to evaluate the post condition. This way of formulating obligations inferred from the static properties of the program under examination and evaluating against a theorem prover shows the benefits of mathematically verifying the expected postcondition against the prevailing preconditions for each program property under examination. The theorem prover partially assists in checking the conditions used to control the logical flow of a program by evaluating the proof

```
public class MyClass1 {
 protected String s1, s2;
 // constructor 1            public void method2() {
 public MyClass1(..) {        String s3;
  s1 = new String(...);        if (cond1) {
 }                              s3 = new String(...);
                               }
 // constructor 2              //...
 public MyClass1(...) {        if (cond2) {
  s1 = new String( ...);       ...s3.length...
  s2 = new String( ... );      }
 }                            }

 public void method1() {
  ... s2.length() ...
 }
```

Figure 1: A null pointer example

obligations formed from the path under examination. A research prototype is under development and the preliminary results are encouraging and demonstrate the feasibility and effectiveness of our approach.

The rest of the paper is organized as follows: Section two briefly discusses the errors in Java that can be detected by our technique and the foundation of our analysis. It also presents the details of our technique applied in detecting null pointer violation in Java programs. Section three discusses the effectiveness of this technique. We conclude in section four, and discuss our future work in section five.

2. Null Pointer Detection

Null Pointer Detection is fundamental to the problem of program analysis. Some of the problems related to detecting null pointer errors are aliasing, name collision, unexecutable paths, uninitialized pointers, Any tool involving automatic program analysis is likely to benefit from information about null pointers in a program. 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). Consider the example inf Fig. 1.

In MyClass1, two instance variables, s1 and s2, are declared. However the programmer forgets to initialize s2 in constructor 1, which is a very common mistake - forgetfulness. The consequence is that the dereferencing of s2 in method1 may cause a null pointer exception at run-time. In method2, the local variable s3 is initialized inside an if statement, and is dereferenced inside a different if statement. The dereference may also cause a null pointer exception at run-time if cond2 is not implied by cond1. These types of errors are very common in programs.

2.1. Foundation of the Analysis

The static analysis is based on the concept of class invariant. A class invariant is a condition satisfied by all non-transient states of the instances of the class. The analysis is based on the following two requirements of class invariants. Assume that CI is an invariant of class C, then

I-1) All public constructors must ensure the invariant, i.e., for public constructor `cntr` of `C`

$$\{\mathit{True}\} \quad \texttt{cntr} \quad \{\mathit{CI}\}$$

I-2) All public methods must preserve the invariant, i.e., for each public method `mthd` of `C`

$$\{\mathit{CI}\} \quad \texttt{mthd} \quad \{\mathit{CI}\}$$

The implication is that for an instance created by any public constructor of the class you may invoke the public methods in any order (assuming the methods are thread safe). Informally, the main steps of detecting errors regarding certain property p are the following:

- Use I-1 to establish a class invariant CIp involving p, preferably as strong as possible.
- Use I-2 to verify that CIp is indeed an invariant, weaken the invariant if necessary.
- Use CIp as the pre-condition of methods to proof certain assertions. We insert assertions regarding p at certain places and try to proof it. If the proof fails, a potential violation is detected.

The detection of null-pointer violations using this technique involves the property isNull(v), which asserts variable v is not null. The analysis consists of the following steps:

1. Establish a class invariant regarding the nullity of reference variables. For each reference variable v, check:

   a) if $\mathit{isNull}(v)$ is a post condition of all public constructors;

   b) if $\mathit{isNull}(v)$ is preserved by all public methods.

   If so, add $\mathit{isNull}(v)$ to the class invariant.

2. Analyze all public methods using the invariant as the pre-condition of public methods. For each dereference of reference variable v, attempt to prove $\neg isNull(v)$.

2.2. Our Approach

Intra class analysis is restricted to information obtained within the class, any intra method analysis is restricted to information obtained within the method. Inter class analysis is restricted to information obtained from all the classes participating in the code to be analyzed, inter method analysis is restricted to information obtained from all the methods participating in the class to be analyzed. In this paper we focus on the intra-method and intra-class analysis.

2.2.1 Determining Class Invariant

To establish a class invariant, we analyze all the constructors of the class. If the reference variable v defined in the class is found to be always initialized through invocation of these constructors, or initialized during static initialization we could consider $\neg isNull(v)$ a potential candidate to be part of class invariants. Assume CI is a set of variables, which can be represented as $\neg isNull(v)$, *CI_candidate* stores a set of candidate variables of CI, which need to be verified. Condition represents the conditions attached to each path. There are three steps to detect a class invariant:

Step 1 : Collect all the not-null static fields into CI. Since the static fields defined in the class will just hold one copy of object through the whole class, they are definitely members of CI.

Step 2 : Check all class's constructors to detect *CI_candidates*. If all constructors initialize the field, we can say it is a candidate of CI.

Step 3 : Check all methods to verify the candidates. If no method changes the candidate field to null, then the field can be a member of CI.

Let us informally define the following:

- *SI_candidate* is static class invariant candidate.
- *SI* is static class invariant,
- *NSI_candidate* is non static class invariant candidate.
- *NSI* is a non static class invariant
- *CI* is class invariant

In order etermine static invariants we need to:

- Check all static fields for not null condition in master static paths, get SI_candidate;
- Check all non static blocks, constructors and methods (static and non static methods) to verify SI_candidate. If no method changes the candidate field to null then the field can be a member of SI.

  $\{SI\_candidate\}$ `non-static-block` $\{SI\_candidate\}$
  $\{SI\_candidate\}$ `constructor` $\{SI\_candidate\}$
  $\{SI\_candidate\}$ `method` $\{SI\_candidate\}$

The determination of non-static invariants has as follows:

- Use SI as precondition for checking all constructors to determine NSI_candidates. If all constructors initialize the field, we can establish it is a candidate of NSI.

  $\{SI\}$ `constructor` $\{NSI\_candidate\}$

- Check all methods to verify *NSI_candidates*. If no method changes the candidate field to null then the field can be a member of NSI.

  $\{NSI\_candidate\}$ `method` $\{NSI\_candidate\}$

After we get the class invariants, we can use them to detect potential null pointer violations. We need to insert assertions $\{\neg isNull(v)\}$ at certain places wherever a dereference of reference variable v is attempted. We attempt to prove these assertions hold with class invariants as the pre-condition of the methods. If the proof fails, a potential violation is detected.

2.2.2 Violation Detection

Program operations that may signal run-time errors during an execution are a starting point in the static debugging process. By inspecting the invariants for the arguments of each program operation we can identify those program operations that can cause run-time errors and flag them for inspection by the programmer. The process for detecting a null pointer violation can be summarized as below: Let:
WC is the weakest pre condition ;
V is any variable that is dereferenced

- Check if:

$$\{WC\} \Rightarrow \neg isNull(v)$$

is not proved , there is a potential null pointer violation.

- Check all master constructor paths if
$$\{SI\} \wedge \{WC\} \Rightarrow \neg isNull(v)$$
is not proved, there is a potential null pointer violation.

- Check all non-static methods, if
$$\{SI\} \wedge \{CI\} \wedge \{WC\} \Rightarrow \neg isNull(v)$$
is not proved, there is a potential null pointer violation.

Appropriate proof obligations are created for each statement in a given path where the field to be proved is dereferenced.

3. The Prototype Tool

The model tree component represents a java class using a logical model. It accepts the output of the java parser and uses it to create an abstract model of the physical structure of the program. The modeling activity depends on an extensive set of classes that provide support for every java programming construct. Name Analyzer gives unique name to each variable declared in the program under examination to avoid name clash resolution.

The model tree is used to create a control flow graph, which creates paths mapping the serial execution of source code. Other pertinent information that is required for analyzing the program and pin pointing those program operations that may cause run-time errors are also attached to this control flow graph. The number of possible paths is a factor of the number of condition statements and their relationships in the program under examination. Inorder to establish all the possible paths for a given program we need to identify all the conditional constructs in the given program that result in defining more paths of execution. Compound conditions need to be broken down into simple conditions, thus defining more paths of execution and helping us pin point the exact sequence of conditions that resulted in a given path. The path generator creates all the possible paths for each control flow graph and serializes them in a file associated to each control flow graph. After identifying all the paths, the analysis will be performed on all the possible paths. Only when the result is correct for all the possible paths, will it be determined to be correct. An overview of our prototype tool is shown in Fig. 2

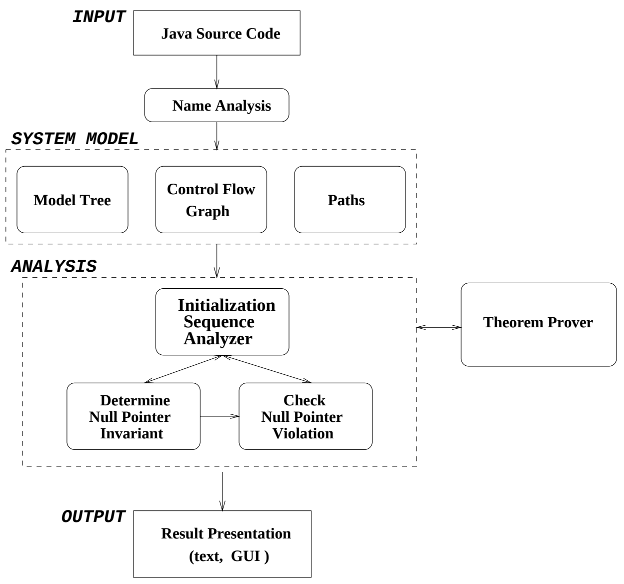
INPUT
Java Source Code
Name Analysis
SYSTEM MODEL
Model Tree
Control Flow Graph
Paths
ANALYSIS
Initialization Sequence Analyzer
Theorem Prover
Determine Null Pointer Invariant
Check Null Pointer Violation
OUTPUT
Result Presentation (text, GUI)

Figure 2: Prototype Components

3.1. Program Model Analyzer

Once the program model is ready, the common properties of the class can be determined by establishing the class invariants. The information obtained from class invariants could be further used to detect potential null pointer violations.

A class in the Java Programming language may contain zero or more static initialization block definitions. The definition of static blocks does play a significant role in the initialization process of the object being instantiated. The determine invariant process does need to know the exact initialization sequence of the object being created to analyze the post-condition of each member variable defined in the class. Static initialization blocks may be static or non static depending on the use of the keyword static. Static and non-static blocks can have branch constructs. In order to track the initialization process of an object, it is important to know the paths through which an object could get initialized. Static blocks can have only static variable declarations. Non static blocks can have both static and non static variable declarations. Static variables get initialized on the first usage of the class. Non static variables get initialized only when an instance of the class is created. The net effect of all the static blocks should be analyzed followed by the net effect of all non static blocks for each individual constructor. This demands the need for identifying all the possible paths as a result of combining all the static blocks. Further there is a need to identify all the possible paths as a result of combining all the non static paths for each constructor defined for an arbitrary class. This is implemented by inferring master static paths for each class and a master constructor paths for each constructor.

3.2. Theorem Prover

For a variable of some reference types we form the predicate $\neg isNull(var)$. This predicate gets modified by replacing the values assigned to the variables as we track each program path bottom up, any conditions are added as preconditions to this predicate. The theorem prover returns a true result if the obligation can be proved to be true which also indicates that the variable may not result in a potential null pointer exception. On the contrary a false result returned by the theorem prover may indicate a potential null pointer exception or the inability of the theorem prover to prove that a potential null pointer exception may not occur.

4. Experiments

A prototype has been developed to evaluate the effectiveness of this technique. We evaluated the effectiveness by giving several different program inputs. The program inputs given considered only the non-distributive nature of the Java programming language. The potential null pointer violations were checked for every variable dereferenced in the given class. Prior to using the prototype, we verified that the program compiles correctly.

We investigate for a variety of different scenarios that may cause a null pointer violation. We have indentified the following possible scenarios and we assigned to each one of them a unique name, which will facilitate the presentation of the coverage of our experiment. The categorization is shown in Fig. 3 where *var* denotes a Java variable of reference type.

Cause Id	Description
C1	class level *var* not initialized
C2	class level *var* becomes null at arbitrary
C3	*var* is dereferenced under two condition blocks
C4	null *var* dereferenced in an unexecutable path
C5	*var* initialized but randomnly becomes null
C6	*var* becomes null before end of iteration of a loop
C7	*var* not initialized but dereferenced is guarded

Figure 3: Categorization of Causes

The cases were classified by the type of variable (static, non-static), scope of the variable (local, class), location of error (static block, non static block, constructor, method). A summary of our classification is shown in Figure 4.

Type	Scope	Location	Cause
static	local	init block	**C1,C2,C3,C4,C5,C6**
		static block	**C1,C2,C3,C4,C5,C6**
		constructor	**C1,C2,C3,C4,C5,C6**
		method	**C1,C2,C3,C4,C5,C6**
	class	init block	**C1,C2,C3,C4,C5,C6,C7**
		static block	**C1,C2,C3,C4,C5,C6,C7**
		constructor	**C1,C2,C3,C4,C5,C6,C7**
		method	**C1,C2,C3,C4,C5,C6,C7**
instance	local	init block	**C1,C2,C3,C4,C5,C6**
		static block	**C1,C2,C3,C4,C5,C6**
		constructor	**C1,C2,C3,C4,C5,C6**
		method	**C1,C2,C3,C4,C5,C6**
	class	init block	**C1,C2,C3,C4,C5,C6,C7**
		static block	**C1,C2,C3,C4,C5,C6,C7**
		constructor	**C1,C2,C3,C4,C5,C6,C7**
		method	**C1,C2,C3,C4,C5,C6,C7**

Figure 4: Potential Error Classification

The same cases were extended further to check potential null pointer violations in inner classes. 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 different faults under different context's until enough samples were collected to evaluate the prototype performance in a meaningful way.

Initial tests conducted were to check for null pointer violations in programs where variables were uninitialized. The program representation takes into consideration the evaluation of initial values of variables as per the Java language specification and is able to identify null pointer violations arising due to uninitialized variables. The invariants help us detect null pointer errors for variables which become null arbitrarily. The prototype is able to detect null pointer violations in statements as well as conditions.

The program representation evaluates more number of paths for compound conditions by splitting it into simple conditions, loops are evaluated as paths with none or more iterations. This simplification helps us identify the exact path and the sequence of statements that will result in a potential null pointer exception. The collection of obligations evaluates any aliases arising during analysis and makes appropriate substitutions in the proof obligations submitted to the theorem prover. In a nutshell it was found that in many cases it was cost effective to use formal program proof techniques as compared to systematic testing. In most of the cases the prototype was successful in identifying invariants and detecting null pointer violation errors. The prototype is free from the side-effects of aliasing. The prototype is also free from pointing errors in unexecutable paths and hence provides a partial solution to the problem of unexecutable paths.

5. Conclusion

The aim of the work presented here is to foster the use of formal methods in specific stages of software development process. We have attempted to make a case for the potential benefits of the use of formal methods without most of its unacceptable costs. The idea of integration of an automatic theorem prover is a major step in this process. Our technique does not require programmers to provide specifications and it is fully automatic. It is capable of detecting null-pointer violations which could result in major cost paying in testing and debugging. The prototype is able to identify a majority of null pointer violation errors based on the information available in an intra-class, intra-method domain. The prototype does not have restrictions on simple control and data flow facilities, it also defines more paths of execution by simplifying branch conditions. This helps the developer segment the paths at a finer level. The tool also addresses issues related to aliasing and is free from pointing errors in those paths that are unexecutable and thus provides a partial solution to the problem of unexecutable paths.

The presented approach identifies the program operations that may signal errors during an execution and describes the sets of erroneous argument values that may cause those errors. The combination of control-flow graph with the theorem prover based approach gives us a mathematical basis to rationalize about correctness. This analysis could be extended to inter-method and inter-class based analysis. The approach could be applied to solve problems related to illegal downcasts and array bounds checks.

In the future, we plan to extend the technique to inter method and inter class analysis and we will handle inheritance and overriding of method, recursive method invocation and cyclic class dependency. We will also apply our technique to detect illegal downcast, array index out of bound.

References

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

[2] D. Evans. *Using Specifications to Check Source Code*, 1994. MS Thesis, Massachusetts Institute of Technology.

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

[4] D. l. Detlefs. An overview of the extended static checking. In *Proc. The First Workshop on Formal Methods inSoftware Practice*, pages 1–9, 1996. ACM-SIGSOFT.

[5] 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.

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

[7] D. Luckham. *Programming with Specifications — An Introduction to Anna, a Language for Specifying Ada Programs.* Springer-Verlag, 1990.

[8] F. Nielson. Semantics-directed program analysis. a toolmaker perspective. In *Proc. SAS96*, pages 2–21. Springer-Verlag, 1996. LNCS Vol. 1145.

[9] A. Rosskopf. Use of a static analysis tool for safety-critical applications. In *Proc. ADA-EUROPE96*, pages 182–197. Springer-Verlag, 1996. LNCS Vol. 1088.

[10] B. Steffen. Property-oriented expansion. In *Proc. SAS96.* Springer-Verlag, 1996. LNCS Vol. 1145.