A Generic Approach of Static Analysis for Detecting Runtime Errors in Java Programs

Xiaoping Jia, 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, sskevoul@cs.depaul.edu

Abstract

This paper presents a generic approach to statically analyze Java programs in order to detect potential errors (bugs). We discuss a framework that supports our approach and carries out the static analysis of Java code automatically. Our approach can automatically detect potential bugs 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. Our approach is distinctive in its emphasis to provide a practical generic mechanism for error detection that is capable of addressing error detection for a variety of error categories via a web of specialized components. A research prototype has been developed that demonstrates the feasibility and effectiveness of our approach.

1. Introduction

Detecting errors in programs has always been one of the most active areas of research [4, 1, 11, 7] in computer science. Numerous research efforts have resulted in techniques that allow us to reason about programs. Concerns about program correctness has led from the Hoare triples [5], to Dijkstra's weakest precondition [1] and currently to complete verification environments such as Higher Order Logic (HOL) [2] and the Prototype Verification System (PVS) [11]. A large number of programming notations exists that use a variety of mechanisms to prevent potential errors from occurring or to notify the user in a safe manner if errors do occur. Examples of such mechanisms include *strong typing*, *smart compilers*, and *runtime checking*. Java [3] is an example of such programming language, which is strongly typed with extensive and sophisticated runtime checking mechanism.

However, even the most sophisticated compilers have a limited ability to detect potential errors. This is due to certain time, resources and performance constraints that limit the amount of checking that compilers can perform. More serious problems, that can cause runtime exceptions, such as attempts to access null pointers, out of bounds array indeces, *etc.* can not be caught by any compiler yet. Most of these kinds of errors are usually caught at runtime.

We approach the problem of potential error detection, using formal methods. We provide a partial solution and a generic mechanism capable of detecting a variety of runtime errors. We apply static analysis and verification techniques, to automatically analyse Java programs and detect potential bugs that can not be detected by compilers and other data flow-based analysis techniques. Our approach is based on the use of the logical concepts of *class invariant* and *weakest precondition* [4]. We developed a *generic* and *deterministic* detection mechanism capable of addressing the problem of error detection for a variety of runtime errors. The detection mechanism comprises a set of generic and specialized algorirhms to carry out analysis of Java code automatically. It consists of the following components:

- *Generic analysis* It can be used to provide the framework for any kind of analysis that we need.
- *Specialized analysis.* It specializes the behavior of some of the steps defined in the generic mechanism.
- *Prototype.* A fully functional prototype implements the algorithms and provides feedback that

is used to enhance and strengthen our approach

Our approach uses a generic mechanism for carrying out the static analysis of Java programs and provides a number of *specialized* components which take into account the idiosycratic details of each category of errors that we are trying to detect.

The detection mechanism of our approach has certain limitations, mainly due to the intractability of theorem proving process. We can not guarantee absolute success in finding *all* bugs 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. The key characteristics and contributions of our approach could be summarized as follows::

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

Complete automation. The entire process of analyzing the source code and reporting the potential violations is fully automatic.

No need for formal specifications. We do not require formal specifications. Indeed, we recover the relevant specifications from the source code.

Flexibility. Our approach can be easily extended to incorporate any user provided specifications which will strengthen the capability of the analysis component and will extend its set of potential errors that can effectively check for.

In the following section we discuss the generic framework and present the algorithms. Section three presents an overview of the prototype tool and its components. Section four discusses the experiments that have been carried out using the prototype tool. Related work and comparisons with our approach is presented in section five. We discuss some future work is section six and conclude in section seven, with a summary of main findings and results.

2. The Analysis Framework

An important concept in our approach is the *invariant* of a class. A class *invariant* is a condition that is satisfied by all non-transient states of the instances of the class. A generic algorithm which can determine an invariant has been developed. It uses a number of generic and specialized components in order to determine invariants related to the specific analysis that is performed. The invariant determination process takes into consideration the different initialization semantics for static and instance variables i.e. static variables are instantiated during the first *active use* [6] of the type (class). The generic algorithms are listed below:

- *DetermineInvariant.* This algorithm is responsible for generating and verifying a class invariant.
- *BreakDownCandInv.* It splits the candidate invariant into predicates involving static variables and predicates involving instance variables.
- *IsInvariant.* It checks whether a predicate regarding class level variables is an invariant.
- *CalculateWP.* (CWP) Given a statement and a predicate, it calculates the weakest precondition.
- *Prove.* It is a call to the thorem prover that attempts to discharge proof obligations.
- *CheckViolation.* It analyzes the implementation of the class and based on the class invariant, it check for potential errors.

Each one of them uses a number of generic and specialized algorithms. Specialized analysis is achieved with a number of specific algorithms that carry out analysis for specific categories of bugs. Some of these algorithms are listed below:

- *ConstructCandInv.* It constructs a potential invariant based on the category of error that we check for
- *Mutate.* The goal of the mutation algorithm 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 can not be mutated anymore and its weakest mutated form is the predicate `true`.
- *CreateTargets* It creates the verification conditions that will ensure the safe use of variables in the Java code.

Determining a class invariant involves the construction of a candidate invariant, its mutation and checking if it satisfies the requirements to be an invariant.

2.1. The DetermineInvariant Algorithm

We informally define the following concepts:

- *Static Invariant* is a condition regarding only static variables of the Java class, that is ensured by all static initialization blocks and is preserved by all constructors and public methods.
- *Instance Invariant* is a condition regarding all class level variables of the Java class, that is ensured by all constructors and is preserved by all public methods.

$\mathbb{P}\,Predicate$ *DetermineInvariant*(**in** *JavaClass Class*)

1. /* Initialization */
   $\mathbb{P}\,Predicate\ Invariant := \varnothing$;
   $\mathbb{P}\,Predicate\ CandInstInv := \varnothing$;
   $\mathbb{P}\,Predicate\ CandStaticInv := \varnothing$;
   $\mathbb{P}\,Predicate\ InstanceInv := \varnothing$;
   $\mathbb{P}\,Predicate\ StaticInv := \varnothing$
   $\mathbb{P}\,Predicate\ PermutCandStaticInv := \varnothing$;
   $\mathbb{P}\,Predicate\ PermutCandInstInv := \varnothing$
   $seq\ Predicate\ MutCandStaticInv := \langle\rangle$;
   seq $Predicate\ MutCandInstInv := \langle\rangle$
2. /* Constructing candidate static and instance invariants*/
   $CandInstInv := Construct\,C\,andinv(Class, CandStaticInv)$;
   $PermCandStaticInv := Breakdown\,C\,andinv(CandStaticInv)$;
   $PermCandInstInv := Breakdown\,Candinv(CandInstInv)$;
3. /* Determine invariant related to static variables */
   for each $pred_{st} \in PermCandStaticInv$ **do**
   $MutCandStaticInv := Mutate(pred_{st})$
   for each mut_{pred} **in** $MutCandStaticInv$ **do**
   if $IsStaticInvariant(mut_{pred})$ **then**
   $StaticInv := StaticInv \cup \{mut_{pred}\}$
   break
   end if
   end for
   end for
4. /* Determine invariant of instance variables */
   for each $pred_{inst} \in PermCandInstInv$ **do**
   $MutCandInstInv := Mutate(pred_{inst})$
   for each mut_{pred} **in** $MutCandInstInv$ **do**
   if $IsInvariant(mut_{pred})$ **then**
   $InstanceInv := InstanceInv \cup \{mut_{pred}\}$
   break
   end if
   end for
   end for
5. /* The invariant found */
   $Invariant := StaticInv \cup InstanceInv$
   return *Invariant*

DetermineInvariant algorithm accepts as input a Java class and returns a predicate that is satisfied by all non transient instances of this class. It reads in the class level variables both static and non static and forms a candidate invariant. It breaks down the formed invariant into two predicates: one that expresses a condition about static variables and one about the all variables. Invariant is broken down to two sets in order to examine seperately the properties that do not depend on the instantiation of the Java class from the ones that do. Each predicate is examined seperately and if it fulfills the requirements of an invariant, is added to the invariant of the class. The output of the algorithm is the invariant for the Java class under examination.

2.1.1. Determining the Invariant

We need to establish a condition (i.e invariant) that is *true* for the relevant variables every time a class is instantiated. The algorithm described below is used to determine invariants related to both static and instance variables.

boolean *IsInvariant*(**in** $\mathbb{P}\,Predicate\ Precond$, **in** $\mathbb{P}\,Predicate\ Candidate$)
boolean $progress := false$
for each $c_i \in InstanceConstructionBlock$ **do**
$progress :=$
$Prove(Precond \Rightarrow CWP(c_i, Candidate))$
if $\neg\ progress$ **then**
break
end if
end for
if $progress$ **then**
for each $m_j \in Methods$ **do**
$progress := Prove(Candidate \Rightarrow CWP(m_j, Candidate))$
if $\neg progress$ **then**
break
end if
end for
return $progress$

Due to the different instantiation semantics between static and instance variables, we actually run the algorithm with different inputs and each time we consider different instantiation blocks. Specifically for the static invariant determination the static blocks must ensure the potential invariant while for instance invariant we consider init blocks and constructors. The precondition that holds before every instantiation of a class is the *StaticInvariant*. In the case that the class has no static variables, the precondition is reduced to the predicate *true*. We use this as a *Precondition* input in our algorithm along with the candidate invariant, *Candidate*. An invariant regarding instance variables has to be ensured by constructors and preserved by each public method.

2.2. The Check Violation Algorithms

The algorithms presented in the previous section determine an invariant with regard to a specific property that we try to ensure. We can now check the usage of each relevant variable to detect any possible violations.

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

In order to do that we provide an algorithm that scans though the implementation of the class, locates all the relevent variables and forms the appropriate verification conditions.

$\mathbb{P}$ *StatementBugs* *Check Class Violation*(**in** *JavaClass Class*, **in** $\mathbb{P}$ *Predicate Invariant*)

1. /* Initialization */
   $\mathbb{P}$ *StatementBugs StatementsWithBugs* := $\varnothing$;
2. /* Check Static Block for potential bugs. Precondition is: **true** */
   StatementsWithBugs := *StatementsWithBugs* ∪ *Check Violation*(*StaticBlock* , *true*)
3. /* Check Constructors for potential Bugs. Precondition is: *StaticInvariant* */
   for each $c_j \in$ *InstanceConstructionBlock* **do**
   StatementsWithBugs := *StatementsWithBugs* ∪ *Check Violation*(c_j , *StaticInvariant*)
   end for
4. /* Check public Methods for potential Bugs. Precondition is: *Invariant* */
   for each $m_j \in$ *Methods* **do**
   StatementsWithBugs := *StatementsWithBugs* ∪ *Check Violation*(m_j , *Invariant*)
   end for
5. /* Return the pstatement with potential violations */
   return *StatementsWithBugs*

In order to identify potential errors, we need to carefully check every statement in the implementation of the class, locate the points that these variables are used, form the appropriate predicate that needs to be *true* in order for the variables to be safely used by the program at the point of execution. This is shown in a general form below:

$\mathbb{P}$ *StatementBugs* *Check Violation*(**in** *JavaBlockCode Block*, **in** $\mathbb{P}$ *Predicate Precondition*)

1. /* Initialization */
   seq Statement ProcessedStatements := $\langle\rangle$;
   boolean *condition* := *true*;
   boolean *correct* := *false*;
   $\mathbb{P}$ *Predicate Targets* := $\varnothing$;
   $\mathbb{P}$ *Variable ProblematicVars* := $\varnothing$;
2. /* Check Java block of code for bugs with a given precondition */
   for each statement $s_i \in$ *Block* **do**
   ProcessedStatements := *ProcessedStatements* $\frown \langle s_i \rangle$
   Targets := *Create Targets*(s_i)
   if *Targets* $\neq \varnothing$ **then**
   for each *pred* $\in$ *Targets* **do**
   correct := *Prove*(*Precondition* ⇒ *CWP*(*ProcessedStatements*, *pred*))
   if ¬ *correct* **then**
   ProblematicVars := *ProblematicVars* ∪ {*var*}
   end if
   end for
   StatementsWithBugs := *StatementsWithBugs* ∪ {s_i → *ProblematicVars*}
   end if
   end for
3. /* Return potential bugs found in the block of code */
   return *StatementsWithBugs*;

3. Prototype Development

The main components of the prototype are listed below:

- **Model constructor** The modeling activity depends on an extensive set of classes that provide support for every Java programming construct, i.e class, method, block of code, statements, types, etc. An abstract syntax tree is created and control flow graph of the program is constructed. Based on the control flow graph model all the paths of execution are identified.
- **Invariant generator** It uses the generic and customizable algorithms to derive an invariant regarding the specific property that is under investigation.
- **Violation detector** Given the invariant from the invariant generator, it scans through class implementation to form the approprate verification conditions, and passes them to the prover.

In general, theorem proving that involves unrestricted predicates is intractable. Our approach has focused on the use of such restricted predicates. Examples of such restricted predicates are the following:

- Checking for illegal dereference of variable of reference type *myvar*. We can formulate the predicate $\neg isNull(myvar)$.
- Checking for array index falling out of bounds while accessing the ith element of *myarray[i]*. We formulate the predicate $i < myarray.length$.

4. Experiments

We assess the capabilities of our approach by conducting a number of small experiments. The results are encouranging and show that the prototype 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 conducted 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. Experimentation is used as a feedback mechanism to our theoretical studies and solutions. The goal is to gain and demonstrate effectiveness, through experimentation. 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. We conducted a large number of tests and the prototype tool exhibited the following key characteristics:

Complete Java coverage. We were able to cover the entire Java language. No syntactical or semantical restriction on the original language has been imposed.

No guarantee of program correctness. Our approach does not promise absolute correctness or absence of errors. It rather strives to identify certain types of potential errors

Application of formal methods Our approach promotes the use of formal methods in a way completely transparent to the user. We can actually apply formal methods with little cost but concrete gains.

Concerete gains. 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 practitioners develop their code.
- It removes the burden of proof from the software practitioners
- The analysis is fully automatic

We have developed a classification scheme for the different causes of errors in both kinds of specialized analysis discussed in this paper: null pointer and array bounds. For each kind of anomaly we analyzed and enumerated the causes and the scenarios under which a violation will be occur in the program. Some of those scenarions are shown in Fig. 1. 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 prototype was capable of finding over 80% of the errors in each scenario. In summary, the early experiments are encouranging. The prototype is still under development and more extensive case studies are planned.

Error	Scenario
null pointer	class level *var* not initialized
array bounds	array initialized with *size*, where *size* ≤ 0
null pointer	class level *var* becomes null at arbitrary
array bounds	array created and accessed in local scope
null pointer	*var* is dereferenced under two condition blocks
array bounds	class level array initialized with `int` constant: a) remains constant b) may change in various constructors
null pointer	null *var* dereferenced in an unexecutable path
array bounds	array access guarded by `arr.length`
null pointer	*var* initialized but randomnly becomes null
null pointer	*var* becomes null before end of loop iteration
null pointer	*var* not initialized but dereferenced is guarded

Figure 1: Experimental Scenarios

5. Related Work

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 are not successful in dealing with invariant properties, pre/post conditions and cannot prevent a number of tedious bugs from occurring. Such bugs include dereferencing a null pointer or array index falling out of bounds, *etc.* These kinds of problems are beyond the capabilities of compilers and subsequently are transfered to the runtime environment where some languages offer exception handling mechanisms.

More recently, research work at DEC laboratories has resulted in *Extended Static Checking* (ESC) [8, 10] and checking object invariants [9]. 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. ESC translates progrmas into a an abstract form based on Dijkstra's guarded command. Their logical framework is untyped first order predicate calculus and the lack of type information for the proofs is covered by the background predicate [9] which includes type axioms. An unsuccessful proof return information about the reason of its failure.

The work presented in this paper is a generalisation and extension of the work [12] on the null pointer detection which carried out by the members of Formal Methods group at Software Engineering Division at DePaul University.

6. Future Work

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

- Stronger analysis techniques which will cover more complicated cases within the already defined categories of potential bugs
- An extensive array of algorithms capable of handling different cases
- Broader coverage of kinds of errors that can be automatically checked. These include but not limited to:
  - illegal downcasting
  - string index out of bounds

7. Conclusion

Usually, checking for errors is not one of the first priorities of a software developer. 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 completeley trasnparent to the software practitioner. By sacrificing guarantee of absolute corectness 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.

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

[2] M. Gordon and T. Melham. *An Introduction to HOL: A theorem proving environment for higher order logic*, 1993.

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

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

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

[6] B. J. Gosling and G. Steele. Java specification language. Technical report, 1996. available from `http://www.javasoft.com`.

[7] D. Jackson. *A formal Specification Language for Detecting Bugs*, 1992. PhD Thesis, Massachusetts Institute of Technology.

[8] 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.

[9] 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.

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

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

[12] S. Sawant. Applying static analysis for detecting null pointers in java programs. Technical report, Oct. 1998. MS Thesis.