

Hypothesis-Based Approach To Detecting Runtime Violations In Java Program Using Automated Theorem Prover

Xiaoping Jia, Lizhang Qin, Hongming Liu

DePaul University, School of Computer Science, Telecommunications and Information Systems
243 So. Wabash Avenue, Chicago IL 60604

Abstract

We develop an approach to apply formal methods to represent the program source codes as a model, after that, using automated theorem prover to try to detect runtime violations in those codes by doing static analysis. Unlike other proof based program verification approaches, this approach is based on hypothesis to develop the formal specification information implied by source codes, such as invariants, preconditions, postconditions and other runtime assertions, then using automated theorem prover to verify the correctness of each statement in the program. Our research work is an extension to compilers, can catch those runtime exceptions which are out of bound of the capability of compiler's control-flow based analysis.

Keywords: Hypothesis, Formal Methods, Automated Theorem Prover, Runtime Violations

1. Introduction

Error detection in programs has always been one of the most active areas of research in computer science. That is because even the elaborate program may have some runtime violations, and those violations may occur on some special conditions during the program is running and may cause severe results in some cases. For example, in 1999, NASA lost its Mars Polar Lander due to a software related problem, and lost \$165 million dollars. People expect that software is error-free, safe and reliable[6].

Currently, software testing by testers is still the No.1 method to find the errors in program, but that is an expensive and unreliable way. In many cases, testers could not test the software thoroughly, so may miss some errors.

Researchers hope to develop some automatic tools, which can find those runtime violations in the software without running it. In the past years, many research have been conducting in order to realize the goal, and gain some fruits.

As we notice, the modern compilers are using control flow based analysis to scan the source codes and report some runtime violations which originally could not be caught by traditional compilers. Although the analysis in those compiler are quite limited due to the lack of reasoning capability and leave the rest of violations to the runtime environment, they show the feasibility and success of applying static analysis to detect runtime violations. Static analysis to source code can be conducted in many other ways than control flow analysis. Another flow based analysis is data flow analysis.

Constructing finite state model for program in order to applying model checking technology to do verification also attracts many researchers, although modeling checking is originally target on hardware design and protocol design and hard to construct such a model without approximation to software.

The approaches using reasoning technology to do logic proof based analysis have the potential to do the modular analysis, which can be applied to variety of software systems. A bunch of

research efforts are in this way, HOL[2], PVS[4], and ESC[5] are some examples. The barrier to applying proof based approaches in industry is that the logic condition information, which are the must to conducting verification process, need to be provided by programmers explicitly for current tools and their underlying approaches.

Comparing the fruits from different approach category, we notice that although proof-based approaches are the only approaches with reasoning capability, they are not so widely used as flow based approaches. There are three main obstacles staying along the way to make proof-based approaches more practical:

Logic information retrieval issue is the first one. Since proof-based approaches need the logic conditions of source codes to conduct the proof process, but in traditional approaches those logic information need to be described by programmers explicitly, then it greatly heavies the burden of practitioners. That is the main difficulty to apply proof-based approach widely. In our approach, we develop those logic conditions implied by source code automatically using an automated theorem prover.

Logic information representation is another obstacle. Those logic condition information need to be represented in some way. One weakness of applying proof-based approach is low expressiveness of their input language. Design By Contract(DBC) is a lightweight formal methods framework for Object-Oriented programs, and DBC has been adapted for a long time, easy to be adapted by users. Although DBC is originally intended to represent the design contract information for the concrete implementation of the codes, we adapt DBC as our formal methods framework with some Z based representation in our research.

Intractability of theorem prover makes thing harder. Theorem prover is the heart to provide reasoning capability in proof-based approach. In order to make verification process automatic, all proof-based approaches need to use an automated theorem prover, but generic automated theorem provers are normally weak to be applied for real-world applications. So in our research, we tailor the needs of proof tasks to our particular domain to design and implement an efficient automated theorem prover.

This paper addresses an new logic proof based approaches, which we called Hypothesis-Based Approach, to do the static analysis to programs targeting on detecting most runtime violations. By discovering the logic information implied by source code according to specific runtime violation category, the whole verification process is fully automatic and does not require programmers to do extra work in order to accommodate our approach and tools.

2. Objective

We introduce an approach based on hypothesis to develop the logic condition information implied by source codes, such as invariants, preconditions, postconditions and other runtime assertions, then using automated theorem prover to verify the correctness of each statement in the program.

In this research project, we choose Java as the target language. We choose Java not only because Java is one of popular Object-Oriented language in current days, but also Java provides a relatively clear and simple language model to be analyzed comparing with some other Object-Oriented languages such as C++, for example, Java[16] lacks arithmetic operations for pointers which may make static analysis process much more complicated.

Also in the research project, we introduce a generic approach and develop a generic framework to detect different categories of runtime violations. But in order to show the feasibility of our approach, we choose null pointer dereference and array access out of bound

exception as the runtime violation categories to conduct research and instrumentation. Noticeable, both of the approach and framework are not restricted by above runtime violation categories, they are extensible and capable of handling other categories. The reason of choosing null pointer dereference and array access out of bound is that they are two kinds of the most common violations in real software and two main causes of breaking safety of software.

One important step to apply formal methods in industry is to introduce some practical solution for the problem to be solved. So in our research, we drop the guarantee of reporting all the potential violations completely, instead, we are providing a practical way to find majority of them in a reasonable time by using reasonable computation resources.

Our approach belongs to logic proof based category to analyze programs, in particular, different from current proof based checkers, our approach does not need programmers to provide additional logic condition information in their source code to conduct the verification process, instead, our approach hypothesizes and discovers those logic condition information implied by source codes according to specific runtime violation category to be scanned, then conducting the reasoning process to verify the program. The whole process is automatic.

Our main goal is trying to find a feasible and practical way to apply formal methods and automated theorem prover to discover logic information from source codes automatically, and provide a generic mechanism to detect the majority of common runtime violations in program automatically without increasing the burden to programmers, and become an useful extension to compiler to improve the quality and reliability of software systems.

Another goal is to adapt formal methods, which is originally basic foundation in computer science and mainly used for research purpose, to practical application in industry. We use Design By Contract with some Z based representation as our formal methods framework.

We also design and implement an efficient and powerful enough automated theorem prover to provide the reasoning capability to our approach.

3. Foundations of Using Logic Reasoning to Verify Java Programs

Before we introduce our new approach, here address the foundations of using logic information to verify program which is already known. This verification process comes from well known Hoare Triple[3] as follows:

$$\{P\} S; \{Q\}$$

where S is a segment of statements

P is the already known preconditions which hold before execution of S

Q is the expected postconditions which to be hold after execution of S

In above triple, the proof obligation to insure the correctness of statement is

$$P \Rightarrow wp(S, Q)$$

where $wp(S, Q)$ is a predefined function. By given a segment of statements S , and a postcondition Q to be hold, $wp(S, Q)$ returns the weakest precondition which need to hold in order to make Q true after executing S

In order to construct proof obligation, there must be P, S, Q and function $wp()$ available, many researches about calculation $wp()$ [2] and some calculation in Java semantics[17][18][19] have been done, code segment S is explicitly given by given the source code, but where are $\{P\}$ and $\{Q\}$ coming from? Existing tools and approaches require programmers to provide those logic

conditions explicitly. Our approach uses reasonable hypothesis and uses automated theorem prover to discover $\{P\}$ and $\{Q\}$ automatically.

In our approach, unless particular mentioning, we choose S from the first statement in the method, to some specific statement in the method body as a composition statement.

4. Hypothesis-Based Approach To Detecting Runtime Violations

Hypothesis-Based approach is using automated theorem prover to discover information and validate program to specific runtime violation category.

Using Hypothesis to Discover Logic Information To Specific Violation Category

Commonly, obtaining the complete logic information for each statement is infeasible unless programmers describe their design and intention for each statement explicitly and thoroughly. But as we mentioned before, the goal of our approach is to detect the *majority* of runtime violations according to *specific* runtime violation category. So the logic information we need are constraint to some particular forms of predicates based on specific violation category, we are not trying to verify the functionality of program implements the designer's idea or not, which is far more than what we expect today.

The general process of using hypothesis to construct logic information contains the following three steps:

- According to specific violation category and logic conditions type be constructed in source code, such as class invariant, method preconditions/postconditions, generalize the form of predicates which might be the logic condition and the rules to verify those predicates.
- According to specific program, construct the hypothesis using the general form.
- Verify the hypothesis to see whether it is a valid logic condition information we need at specific point in the program.

Figure 1 illustrates the components used in our approach and the interaction/relationship between them. We have three basic components in our approach, *Hypothesis Generator*, *Hypothesis Verifier* and *Static Analyzer*.

Hypothesis Generator will use some heuristic mechanism to generate the logic condition hypotheses, for example, class invariants, methods preconditions/postconditions, etc. Those predicates are not randomly chosen, they have to show a reasonable probability to be real logic conditions which are implied by source code.

Hypothesis Verifier will filter all the hypotheses generated by *Hypothesis Generator* using some specific rules, only output the valid logic condition information which are consistent with source codes. Those logic condition information are required by programmer in other proof based approaches.

Static Analyzer accepts both the source code and the logic condition information from *Hypothesis Verifier*, then use automated theorem prover to check the validity of each statement according to specific runtime violation category.

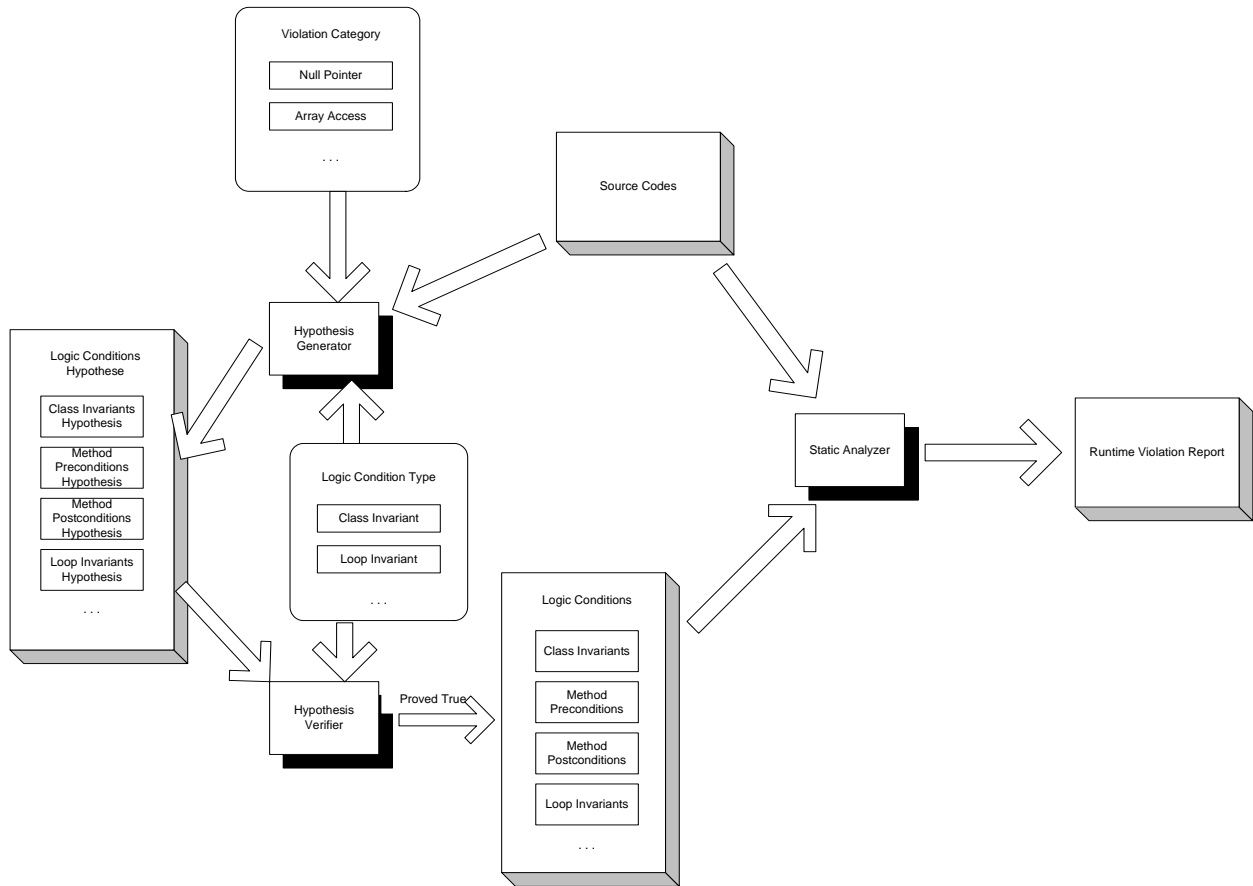


Figure 1. Basic Components In Hypothesis-Based Approach

Figure 1 also shows the basic factors to impact hypothesis heuristic mechanism. These hypotheses are essentially depending on the violation category we are interested in, for example, for null pointer exception, the basic element in hypothesis is in form of

$obj \neq null$

where obj is an object variable.

Besides that, the logic condition type also affects the heuristic algorithm. For example, we are using different ways to construct class invariants and loop invariants. Furthermore, the logic condition type also determine the algorithms(rules) to validate hypotheses. In the rest of this section, we illustrate how to combine those factors to make heuristic hypothesis by addressing some instances.

This process can be applied to discover different kinds of logic condition information implied by source codes. Hypothesis-based approach is a general approach, we are not restricted by only applying it the following uses, but can extend its use in the future research and improve the capabilities and accuracy of tools.

An Example Class

Before we illustrate the current algorithm to apply Hypothesis-Based Approach to discover the logical condition information and detect the runtime violations according to different runtime violation category, in Figure 2, we shows an example Java class which contains both potential null pointer and potential array access out of bound violations.

In the rest of this paper, we will use the source codes shown in *Tools* class to give a concrete example to show the process of analysis.

```
1:  public class Tools {
2:      String title=null;
3:      String name=null;
4:
5:      public Tools() {
6:          title="Default Title";
7:          name="Default Name";
8:      }
9:
10:     public void changeTitle(String s) {
11:         title=s;
12:     }
13:
14:     public void displayTitle() {
15:         System.out.println(title+" with length "+title.length());
16:     }
17:
18:     public void displayNameAndTitle() {
19:         System.out.println(name+" with length "+name.length());
20:         System.out.println(title+" with length "+title.length());
21:     }
22:
23:     public void sort(int[] arr) {
24:         int i = arr.length;
25:         while (i > 0) {
26:             int j = 0;
27:             while (j < i) {
28:                 if (arr[j] > arr[j + 1]) {
29:                     swap(arr, j, j+1);
30:                 }
31:
32:                 j = j + 1;
33:             }
34:             i = i - 1;
35:         }
36:     }
37:
38:     public void swap(int[] arr, int p1, int p2) {
39:         int temp;
40:
41:         temp=arr[p1];
42:         arr[p1]=arr[p2];
43:         arr[p2]=temp;
44:     }
45: }
```

Figure 2. An Example Java Class with Runtime Violations

Construct Assertions to Specific Violation Category

We can assert the postconditions for a statement by looking at the next statement and the specific violation category. For the following statements structure:

$S; \{Q\} S'$

where S is a statement, may be composition statement.

Q is the expected postconditions which to be hold after execution of S
 S' is a single statement next to S

S is followed by S' , and according to specific violation category, we can construct some assertions(preconditions for S') in order to make S' valid. We hypothesize those assertions as $\{Q\}$, the postconditions which are expected to be hold after execution of S .

Those $\{Q\}$ don't need to be verified, they are the predicates which must be hold, otherwise S' will cause runtime violation in some execution cases.

Currently we are dealing with two categories of runtime violations

- Invalid dereference for pointers, we also call it null pointer exception.
- Invalid access array index which is out of bound, we also call it array bound exception.

These are two most common types of violations which can not be caught by compiler, only caught at runtime by JVM.

For null-pointer, if the statement S' contains any expression in the following form,

$v.m(\dots)$

where v is an object variable;

m is a method which can be applied to v

We construct $\{Q\}$ by using

$v \neq \text{null}$

Otherwise $\{Q\}$ is simply true.

For example, in Line 20 of *Tools* class in Figure 2, there is a dereference for class variable *title*, then we construct $\{\text{title} \neq \text{null}\}$ as the postcondition to be hold for statement in Line 19.

For array bound checking, if the statement S' contains any expression in the following form,

$arr[i]$

where i is an integer variable

arr is an array object.

We construct $\{Q\}$ by using

$i \geq 0 \wedge i < arr.length$

Otherwise $\{Q\}$ is simply true.

For example, in Line 43 of *Tools* class in Figure 2, there is an array access $arr[p2]$, then we construct $\{p2 \geq 0 \wedge p2 < arr.length\}$

For the code segment which ends with the last statement in method, we simply set its postcondition to be true.

Constructing Precondition Hypotheses for Statements

Here we use the class invariants to be the preconditions for each public method in the class[8]. An invariant of a class is a formally specified condition that always holds on any object of that class whenever it is in a stable state. Since class invariants will hold before and after the execution of each public method in the class, they are the preconditions for those public methods, although they are not the only preconditions.

We generate some hypotheses for each violation category, then try to prove them. If some hypothesis can be proved true, then it is part of the class invariants, and also part of the preconditions of public methods.

One of key to determine the class invariants is how to generate hypotheses[8].

For null pointer checking, for each object field *obj* in the class, we construct the hypotheses like

obj != null

For example, for *Tools* class shown in Figure 2, we hypothesize the class invariants are
{ *title* != null \wedge *name* != null }

For array bound checking, we are interested in a predicate regarding the size of the array that can be used as invariant. For array type of field *arr* with length *arr.length* and initial size, we are trying to formulate a hypothesis in the form

arr.length >= size

There are three ways to get the size of array as follows, assume *arr* and *brr* are two arrays.

- *arr* = { *e*₁, *e*₂, ..., *e*_{*n*} } is interpreted as *arr.length*=*n*;
- *arr* = *brr* is interpreted as *arr.length* = *brr.length*;
- *arr* = new Type[*size*] is interpreted as *arr.length* = *size*;

For example, for *Tools* class shown in Figure 2, since there is no field with array type, then the class invariants are simply { *true* }

Proving Hypotheses to Preconditions

Give the following form of class structure:

```

class ClassName {
    ClassName(...) { C1 }
    ClassName(...) { C2 }
    .....
    ClassName(...) { Cn }

    public type1 method1(...) { M1 }
    public type2 method2(...) { M2 }
    .....
    public type2 methodm(...) { Mm }
}

```

And the following notations

- CP*: The conditions to be hold before running each constructor, normally generated by initializer of the class.
- C_i*: The *i*th constructor, *C*₁, *C*₂, ... *C*_{*n*} are the constructors for that class.
- H_j*: The *j*th hypothesis for class invariants
- M_k*: The body of *k*th method in class, *M*₁, *M*₂, ... *M*_{*m*} are the methods for that class.

For class invariant hypothesis *H_j*, if we can prove

$\forall i: 1..n \bullet CP \Rightarrow wp(C_i, H_j)$

$\forall k: 1..m \bullet CIC_j \Rightarrow wp(M_k, H_j)$

Then *H_j* is an invariant for that class, and precondition for each public method.

After getting the invariants for the class, we can use them as the preconditions for each public method, and doing the proving process for each statement by calculating the weakest precondition from the assertion to the top of the method.

Handling the iteration structure in program

Besides the above two use of hypothesis to get logic information, when we calculate wp() function for iteration structure, basically while loop in Java, it is still important to discover the

loop invariants. Otherwise, we might be forced to simplify the loops to iterate certain times, which is not accurate and is the cause of some false-positive results.

We use hypothesis to get loop invariants, and use those loop invariants to conduct calculating of $wp()$ for while loop statement.

Following structure shows the basic logical structure for a while loop

```
{P}
while (C) {
    {wp(S, I)}
    S;
    {I}
}
{Q}
```

where S is the while loop body.

P is the already known preconditions which hold before execution of while loop

Q is the expected postconditions which to be hold after execution of while loop

I is the loop invariants, which to be hold after each iteration.

We use

$$wp(\text{while}, Q) = [C \wedge wp(S, I)] \vee [\neg C \wedge Q]$$

to calculate the weakest precondition for the whole while loop.

Here, the loop invariants $\{I\}$ is unknown, unless they are specified by the programmers. We are going to construct the loop invariant hypotheses. Generally, we can use $\{Q\}$, $wp(S, Q)$ as possible hypotheses. For more accurate hypothesis, we need to do more test and observation depending on the while loop patterns.

If H is a loop invariant hypothesis, then use the following rules to verify whether it is a loop invariant or not.

$$H \wedge C \Rightarrow wp(S, H)$$

$$H \wedge \neg C \Rightarrow Q$$

where H is the hypothesized loop invariant.

If above predicates are proved as theorem, then H is a loop invariant.

Although the above verification process for $\{I\}$ is somehow strong than necessary, if a hypothesis passes the above test then it must be a loop invariant. If the above process fails, we can still use the approximation model to limit the loop to iterate certain times.

5. Experiments

In order to show the feasibility and make evaluation to our approach, we have developed a prototype, which can provide:

- **A generic detection mechanism**

We design and implement a generic framework for program analysis, which does common task based on our approach and leave the parts which need to be implemented differently according to specific violation category for further development. This mechanism provides an efficient and effective way for a variety of violation detection.

- **Special analysis mechanism for null pointer and array access out of bound**

We implement the analysis mechanism for the above two violation categories, using the hypothesis discussed in section 3.

- **Complete Java coverage**

Neither syntax nor semantic restrictions have been imposed on the original program.

- **A automated theorem prover**
A powerful enough automated theorem prover is integrated in our prototype, which combines with some algebra calculation capability to improve efficiency of proving process.
- **Complete automation**
The entire process of analyzing source codes is fully automatic. No extra efforts are needed for programmers to conduct the analyzing process.
- **Side effect removing mechanism**
Since Java semantics include some operation with side effects, they have to be eliminated before conducting analysis, because the basic calculation process for `wp()` only accepts the simple calculation expressions. For example,

$$i=j++;$$
is a valid Java statement with side effects. In our prototype, a module is developed to transform the original source codes into the equivalent codes without any side effect, this module is executed before static analysis process.

In this early stage of prototype, there are some limitations:

- **Only analyzing intra-methods codes.** Current prototype could not analyze the inter-methods or inter-class method invocation, but that is not a limitation of approach, we are going to enhance the prototype to remove this restriction very soon.
- **Not implementing loop invariants based calculation of `wp()` for iteration.** In the existing prototype, a simplified model for iteration is used, in detail, a loop iteration has been expanded to several different paths based on the iteration times.
- **Using a relatively strong form of hypothesis candidates.** The prototype does not match the pattern of source code in order to make more reasonable and weaker form of hypothesis candidates, only hypothesize according the runtime violation category.

Example of Analysis Result

Here we show the analysis result got from our prototype to analyze *Tools* class shown in Figure 2.

a. Null Pointer Analysis

For the *Tools* class in Figure 2, when we are interested in proving the correctness of all dereference, according to our approach, we hypothesize the preconditions for each methods as follows based on the class field:

$$title \neq null$$

$$name \neq null$$

Since *name* is initialized by the only constructor, and every method in class does not nullify that field, then $\{ name \neq null \}$ is a class invariant and precondition for each public method.

Although *title* is initialized by the constructor too, $\{ title \neq null \}$ is not a valid class invariant, because for *changeTitle(String s)* method, according to our approach, we construct the following proof obligation to check whether *title* \neq null is an invariant or not.

$$title \neq null \Rightarrow wp(title=s; title \neq null)$$

and we have

$$wp(title=s; title \neq null) = (s \neq null)$$

Then we send

$$title \neq null \Rightarrow s \neq null$$

to automated theorem prover, and get response that it is not a theorem.

Figure 3 shows the results of analysis for null pointer.

Class Invariant Candidates	<i>title != null && name != null</i>		
Class Invariant	<i>name != null</i>		
Violations found by our prototype	1	Variable	<i>Title</i>
		Position	Line 15
		Assertion to proof	<i>title != null</i>
		Statement	<i>System.out.println(title+" with length "+title.length());</i>
	2	Variable	<i>Arr</i>
		Position	Line 24
		Assertion to proof	<i>arr != null</i>
		Statement	<i>int i = arr.length;</i>
	3	Variable	<i>Arr</i>
		Position	Line 28
		Assertion to proof	<i>arr != null</i>
		Statement	<i>if (arr[j] > arr[j + 1])</i>
	4	Variable	<i>Arr</i>
		Position	Line 41
		Assertion to proof	<i>arr != null</i>
		Statement	<i>temp=arr[p1];</i>
	5	Variable	<i>Arr</i>
		Position	Line 42
		Assertion to proof	<i>arr != null</i>
		Statement	<i>arr[p1]=arr[p2];</i>
	6	Variable	<i>Arr</i>
		Position	Line 43
		Assertion to proof	<i>arr != null</i>
		Statement	<i>arr[p2]=temp;</i>

Figure 3. Null Pointer Analysis Result for Tools Class

All the errors reported in Figure 3. are not easily detected by people. In Line 15, programmer may intuitively believe *title != null* when doing dereference, but if client codes invoke *changeTitle()* first with a null pointer parameter, then Line 15 will result in a null pointer exception during runtime. Errors in Line 41 to 43 are even more tricky. When *swap()* is used by *sort()* method in the same class, we know *arr != null*. But since *swap()* is a public method, it can also be invoked by any other clients, so no one can guarantee the formal parameter *arr* in *swap()* is always an not null object. Errors in *sort()* is for the same reason.

b. Array Access Analysis

For the *Tools* class in Figure 2, when we are interested in proving the correctness of all array access, according to our approach, we hypothesize the preconditions for each methods based on the class field, but there is no array field, then the class invariant candidate is simply *true*,

This candidate *true* is the class invariant and precondition for each method.

Figure 4 shows the results of analysis for array access.

Class Invariant Candidates	<i>True</i>		
Class Invariant	<i>True</i>		
Violations found by our prototype	1	Variable	<i>Arr</i>
		Position	Line 28
		Assertion to proof	$j \geq 0 \ \&\& \ j < arr.length \ \&\& \ j+1 \geq 0 \ \&\& \ j+1 < arr.length$
		Statement	<i>if (arr[j] > arr[j + 1]) {</i>
	2	Variable	<i>Arr</i>
		Position	Line 41
		Assertion to proof	$p1 \geq 0 \ \&\& \ p1 < arr.length$
		Statement	<i>Temp=arr[p1];</i>
	3	Variable	<i>Arr</i>
		Position	Line 42
		Assertion to proof	$p1 \geq 0 \ \&\& \ p1 < arr.length \ \&\& \ p2 \geq 0 \ \&\& \ p2 <= arr.length$
		Statement	<i>arr[p1]=arr[p2];</i>
	4	Variable	<i>Arr</i>
		Position	Line 43
		Assertion to proof	$p2 \geq 0 \ \&\& \ p2 < arr.length$
		Statement	<i>arr[p2]=temp;</i>

Figure 4. Array Access Analysis Result for Tools Class

All the errors reported in Figure 4. are not easily detected by people. *sort()* method tries to implement a bubble sort algorithm, but not using the correct assignment in Line 24, which should be *int i=arr.length-1*, not *int i=arr.length* in example. Then definitely in Line 28, *arr[j+1]* will throw an array access of out bound exception when $i=arr.length$ and $j=arr.length-1$. Errors in Line 41 to 43 are even more tricky too. When *swap()* is used by *sort()* method in the same class, we know *p1* and *p2* will be in the valid range for *arr*. But since *swap()* is a public method, it can also be invoked by any other clients, so no one can guarantee the access of *arr[p1]* and *arr[p2]* are always valid.

Experiment Results

Although the limitations exist in current prototype, we get encouraging results when applying prototype to analyze around 200 small Java programs, which cover the majority of common faults in real programs. Currently, for those small java programs, our prototype can detect 100% of the possible runtime violations without false-positive results. Those test cases cover the following scenarios.

- Null Pointer Analysis
 - Lack of initialization
 - Variable initialized but is nullified at arbitrary point.
- Array Access Analysis
 - Invalid array initialization
 - Array is created and illegal accessed within local scope
 - Array declared as class or instance variable and accessed

In summary, although those elaborated test cases could not reflect the complexity of real software, the early experiments show the feasibility and effectiveness of our approach. The prototype is still under development and more extensive case studies are planned.

Factors to affect accuracy of analysis

After analyzing the experiment result, we found that the following factors in prototype will impact the accuracy of analysis:

- **Proper Hypotheses**
Because our approach depends on hypothesis to discover logic information implied by source codes, the first step of analysis is to construct proper hypotheses. If the hypotheses are not reasonable, they will fail the test according to specific verification rule. Therefore, building reasonable hypotheses is important to our approach. Some false-positive results from our experiment are coming from too strong hypotheses.
- **Proper Verification Rules**
The verification rules for hypotheses when we discover the preconditions of methods are theoretically correct. But verification rules discussed in section 3 for loop invariant is more strong than necessary, so it may report the real loop invariants as non-invariants. So more accurate proven rules we have, more accurate analysis results we will get.
- **Automated Theorem Prover**
Automated theorem prover is an intractable problem, we don't expect a full functional automated theorem prover, instead, we integrate an efficient one with reasonable capability in our prototype. Even though, a more powerfully automated theorem prover can improve the capability of analyzer.

6. Related Work

Flow-Based Approaches

Control flow based analyses are extensively used in compiler's optimization process and now are enhanced to do some runtime violation checking. For example, the modern compiler of Java, can provide early detection of a various anomalies, such as a local variable is not instantiated before its usage. Control flow analysis is lack of reasoning capabilities, which are necessary in order to detect a variety of potential runtime violations.

Furthermore, many research works are undergoing to use another type of flow analysis, which is data flow based analysis. The first analyzer "lint" of this kind was developed for the C language. Then, PC-lint/FlexeLint2, QA C, QA C++3, LCLint[9] and others were developed. They are based on a fast data flow analysis of a program in which intra-procedural analysis is used together with elements of inter-procedural analysis; in addition, the constant propagation is often implemented. The main drawback of those tools is that they generate a long list of warnings instead of reporting runtime violations. OSA[10] is deploying a context-sensitive data flow based analysis to source code in order to eliminate those warnings and give customers more accurate violation reports, WASP is the commercial software using the research fruits of OSA and analyze the source codes written in Java. By using data flow analysis, the checkers need to evaluate the value of variables and conditions, because of this restriction, it is hard to detect the runtime violations in the source code in which the condition and values are coupled with users interactions. Also those tools are seldom used to scan the library codes without knowing the clients. Those runtime violations are still be preserved until runtime and become the runtime exceptions and cause the program to terminate abnormally.

The advantage of our approach with comparison to flow based approach is that our approach has the reasoning capability, so can do modular checking. That is, our approach checks each class individually. This means that our tools can be applied to the code that calls libraries even if the code for the libraries is not available. It also means that our tools can be applied to library code whose clients or subclasses have not yet been written.

Model Checking Based Approaches

Model checking based analysis mainly focuses on the temporal properties of a system and is successfully applied in verification of hardware design and protocol design[11][12]. Since the last decade, many researches are conducted to apply model checking to software verification. JavaPath Finder[13], Bandera[14], and SLAM[15] from Microsoft are some examples. Two main problems which come with applying model checking to software are the complexity of state and dynamic nature of most programs. So in many cases, building a finite-state model can only be done by generating an approximate model from the original codes, even though, the extremely huge model requires tons of memory and computation resources.

As one member of logic proof approaches to do program analysis, our approach does not need to construct the whole finite state model comparing with model checking based approach, and does not need huge memory and computation resources to conduct the verification process.

Proof Based Approaches

Using logic to prove predicates has a relatively long history in Computer Science study, also many former research results push researchers to think about using logic proof to reason about programs. Those effects started from Hoare triples[3], to Dijkstra's weakest precondition[1], and turns into the complete verification environments, such as Higher Order Logic(HOL) [2] and the Prototype Verification System(PVS)[4]. More recently, COMPAQ research lab(formerly DEC research lab) build their Extended Static Checking(ESC) system[5], which is originally target on Modula-3, and now extended to Java program. 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.

The main weakness of existing proof based approaches is that they requires the programmers to provide logic condition information as annotations in order to carry out the analysis. For example, ESC includes an annotation language with which programmer can express design decisions using light-weight specifications, then ESC uses the given annotations to checks that whether the program is consistent with them or not. As the same effect by some data flow based tools which depend on programmer defined pragmas to conduct analysis, such as LCLint, current proof based checkers greatly limit their use, caused by that weakness.

The main contribution of our approach to traditional logic proof based approach, such as ESC, is that our approach not only use the reasoning technology to verify source code, but also use automated theorem prover to discover logic information implied by program based on proper hypotheses according to specific runtime violation category. This means our approach breaks through the limitation of other proof based approach, which requires programmers to provide their design and logic condition information explicitly. Our approach provides a generic way to get those information automatically when we focus on specific runtime violation categories.

7. Conclusion and Future Work

In our research, we not only address a new approach to conduct a special kind of program verification, which is to automatically detect the specific category of runtime violations, but also show the feasibility of using formal methods to discover the logic condition information implied by source codes.

Since this is still a research project undergoing, in this early stage of prototype, there are some limitations:

- **Only analyzing intra-methods codes. C.**
- **Not implementing loop invariants based calculation of wp() for iteration.**
- **Using a relatively strong form of hypothesis candidates.**

In the future research work, we have two main directions:

- Stronger analysis based on more application of hypothesis and proving process. We are going to make more accurate hypothesis not only on violation category, but also on the pattern of the source codes, which will discover the weaker form of preconditions or invariants and provide more accurate result of analysis.
- Optimization of analysis process. In order to make our tools more practical, we need to deal with the path explosion problem, our future research may focus on the path reduction and analysis optimization, also continue to develop new heuristics, decision procedures and tactics that will extend the capabilities and efficiency of our automated theorem prover.

References

- [1] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation Fixpoints, in Proc. ACM SIGPLAN Conference on Programming Languages, pp 238-252, 1977
- [2] E. Dijkstra, Guarded Commands, Nondeterminacy and Formal Derivation of Program. Communications of ACM, 18(8), pp 453-458, 1975
- [3] A. Hoare, An Axiomatic Basis for Computer Programming. Communications of ACM, 12(10), pp576-580, 1969
- [4] D. Kemp, G. Goodfellow, The Official Report, technical report, ACM SIGSOFT, 1990. Software Engineering Notes.
- [5] K.R.M Leino, R. Stata, Checking Object Invariants, technical report, Digital Equipment Corporation Research Center, 1997. Palo Alto, CA.
- [6] J. M. Schumann, Automated Theorem Proving in Software Engineering, Springer 2001
- [7] David Gries. The Science of Programming. Springer-Verlag, 1981
- [8] S. Skevoulis, X. Jia, Generic Invariant-Based Static Analysis Tool For Detection of Runtime Errors in Java Programs, 2000
- [9] D. Evans, Static Detection of Dynamic Memory Errors. Proc. ACM SIGPLAN'96 Conf. on Prog. Lang. Design and Implem., SIGPLAN Notices, 31(5): pp 44-53, 1996.
- [10] V. I. Shelekhov, S. V. Kuksenko, A.P.Ershov, On the Practical Static Checker of Semantic Run-time Errors, Institute of Informatics Systems, Siberian Division, Russian Academy of Sciences
- [11] K. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer, 1993.
- [12] G. J. Holzmann, Design and Validation of Computer Protocols. Prentice Hall, 1991

- [13] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking Programs. In 15th Conference on automated Software Engineering(ASE), pp 3-12. IEEE Press, 2000
- [14] C. S. Pasareanu, M. B. Dwyer and W. Visser, Finding Feasible Counter-examples when Model Checking Java Programs, Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Springer-Verlag, April, 2001.
- [15] T. Ball, S. K. Rajamani, Automatically Validating Temporal Safety Properties of Interface, SPIN 2001, Workshop on Model Checking of Software, LNCS 2057, May 2001
- [16] J. Gosling, B. Joy, and G. Steele. The Java[™] Language Specification. Addison-Wesley, 1996.
- [17] B. Jacobs, E. Poll, A Logic for the Java Modeling Language JML
- [18] B. Jacobs, Weakest Precondition Reasoning for Java Programs with JML Annotations.
- [19] S. Skevoulis, A Light-weight Approach to Applying Formal Methods in Software Development, technical Report, DePaul University, 1999