A Hypothesis-Based Approach to Detecting Runtime Violations

Lizhang Qin, Xiaoping Jia, Hongming Liu

School of Computer Science, Telecommunication and Information Systems
DePaul University
243 Wabash Ave, Chicago, IL, USA

Abstract. We have developed an approach to apply formal methods to represent program source code as a model and use an automated theorem prover to detect runtime violations by doing static analysis. Unlike other proof-based program verification approaches, this approach is based on a hypothesis to develop the implicit specification information, such as invariants, preconditions, postconditions, then using an automated theorem prover to verify the correctness of each statement in the program. Our research work can catch those runtime exceptions that are beyond the capability of control flow-based analysis in compilers.

1 Introduction and Objective

Error detection in programs has always been one of the most active areas of research in computer science. The reason is that people expect software to be error-free, safe and reliable [6], but even an elaborate program may have runtime violations, potentially causing severe results. Currently, software testing is still the No.1 method to find the errors in program, but it is an expensive and unreliable process. Researchers hope to develop some automatic tools, which can find those runtime violations without running the program. In the past, many researchers have been attempting to realize this goal, and some progress has been made. Modern compilers use control flow-based analysis to scan the source code and report some runtime violations which originally could not be caught by traditional compilers. Although the analysis in those compilers are quite limited due to the lack of reasoning capability and leave the majority of violations to the runtime environment, they show the feasibility and potential success of applying static analysis to detect runtime violations. Using model checking technology to do software verification also attracts many researchers. The approaches that use reasoning technology to do logic proof based analysis have the potential to perform modular analysis, which can be applied to a variety of software systems. Many research efforts follow this strategy, including HOL [2], PVS [4], and ESC [5]. The problem of applying the existing proof-based tools in industry is that the program specification, which is necessary to the verification process, needs to be explicitly provided by programmers.

This paper addresses a new logic proof-based approach, which we call the Hypothesis-Based Approach, to do static analysis of programs as a way of detecting most runtime violations. By discovering the implicit program specification, the verification process as a whole is fully automatic and does not require programmers to do extra work to accommodate our approach and tools.

We introduce an approach based on the generation of hypotheses to develop the implicit program specification, such as class invariants, method preconditions, postconditions, then use an automated theorem prover to verify the correctness of each statement in the program.

In this research project, we chose Java [13] as the target language and chose null pointer dereference and array access out of bound exceptions as the runtime violation categories to conduct research and instrumentation. Notice that both the approach and tools are not restricted to these runtime violation categories; both are extensible and capable of handling other categories. One important step in the application of formal methods in industry is to introduce some practical solution for the problem to be solved. In our research, we remove the guarantee that we will report all of the potential violations, instead we provide a practical way to find majority of them in a reasonable time and using reasonable computation resources.

2 Hypothesis Approach

The basic software verification process comes from the well-known Hoare Triple [3]. Obtaining the complete program specification for each statement is not feasible unless programmers describe their design and intention explicitly and thoroughly. But as mentioned before, the goal of our approach is to detect the *majority* of runtime violations within a *specific* runtime violation category. The program specification we need are constrained to some particular forms of predicates. We do not attempt to verify the functionality of a program.

The process of using hypotheses to construct specification contains the following three steps: according to the specific violation category, generalize the form of predicates which might be the candidates and the rules to verify those candidates; for the specific program, construct hypothesises using the general form; verify the hypothesis.

Fig. 1 illustrates the components in our approach. *Hypothesis Generator* uses a heuristic mechanism to generate the logic condition hypotheses. These predicates are not randomly chosen, they have to show a reasonable probability to be real program specification. *Hypothesis Verifier* filters all the hypotheses generated by *Hypothesis Generator* using some specific rules in order to output the valid program specification that are consistent with the source code. *Static Analyzer* accepts both the source code and the program specification from *Hypothesis Verifier*, uses the automated theorem prover to check the validity of each statement.

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

obj != null   where *obj* is an object variable.

In addition, the logic condition type also affects the heuristic algorithm. For example, we are using different methods to construct class invariants and loop invariants. The logic condition type also determines the algorithms used to validate the hypotheses. In the rest of this section, we illustrate how to combine those factors to create a heuristic hypothesis

A hypothesis-based approach is a general approach, we are not restricted to only the following applications and can extend its use in the future research

2.1 Constructing Assertions

We assert the postconditions for a statement by looking at the next statement. For

S; *{Q}* S'

where S, S' are statements; *Q* is the expected postconditions after execution of S;

We construct some assertions (preconditions for S') in order to make S' valid. We hypothesize those assertions as *{Q}*, the postconditions after execution of S.

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

v.m(….) where *v* is an object variable; *m* is an method which can applied to v

We construct *{Q}* by using *{v != null}*, otherwise *{Q}* is simply true.

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 $\{\, i>=0 \wedge i < arr.length \,\}$, otherwise {Q} is simply true.

2.2 Constructing Precondition

We hypothesize some invariants for each violation category. If a hypothesis can be proven to be true, then it is part of the class invariants, and also part of the preconditions of public methods.

One key to determining the class invariants is the generation of hypotheses [7].

For null pointer, for each object field *obj* in the class, we construct the hypotheses *obj* != null .

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

2.3 Proving Hypotheses

Given the following form of class structure:

`class` C { $C_1, C_2, \ldots C_n, M_1, M_2, \ldots, M_m$}

`where` C_i is the *i*th constructor; M_k is *k*th method in class

CP: The conditions to be held before running each constructor.
H_j: The *j*th hypothesis for class invariants

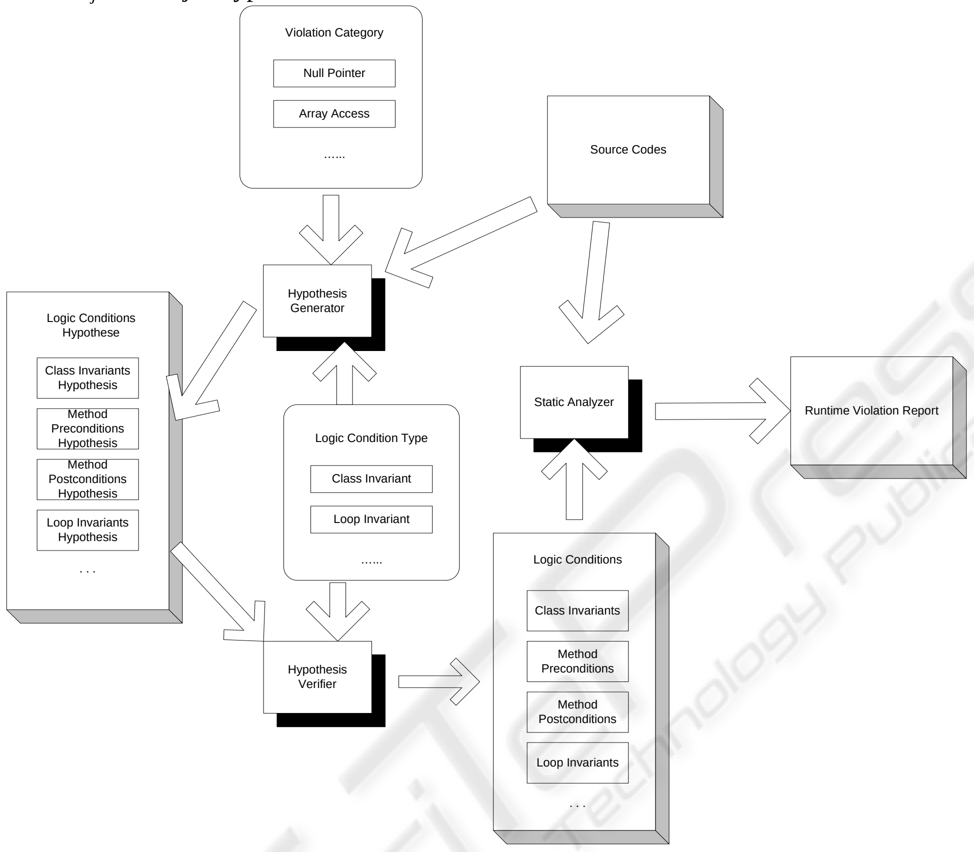

Fig. 1. Basic Components In Hypothesis-Based Approach

If $\forall i$: 1..n $\bullet$ $CP \Rightarrow \text{wp}(C_i, H_j)$ $\wedge$ $\forall k$: 1..m $\bullet$ $CIC_j \Rightarrow \text{wp}(M_k, H_j)$

Then H_j is an invariant for that class and a precondition for each public method. After getting the invariants for the class, we can use them as the preconditions for each public method, and begin the proving process for each statement by calculating the weakest precondition from the assertion to the top of the method.

3 Experiments

In order to show the project feasibility and evaluate our approach, we have developed a prototype, which includes the special analysis mechanism for null pointer and array access out of bound, the complete Java coverage, an automated theorem prover, the side effect eliminating mechanism and the inter-method and inter-class invocation analysis.

As described before, the intractability of Automated Theorem Prover is an important issue to solve. We integrated symbolic algebra calculation as part of our automated theorem prover, which greatly reduces the burden of proving process. Also we use a special prove strategy to handle the specific type of predicate we construct, for example *v != null.* According to our test, this theorem prover can provide an effective mechanism to prove predicate in an efficient way.

Although limitations exist in the current prototype, we receive encouraging results when applying the prototype to analyze around 200 small test Java programs, which are made manually. For those test programs, our prototype generate a 100% correctness on the analysis, that shows the current prototype does exactly what we expect.

In order to evaluate the real capability of our prototype and approach, we develop a mutation mechanism to generate test cases automatically. The current mutation mechanism take the java source code as input, then scan all the "*new*" statement, remove each once to output a mutated version. By this way, we simulate the common fault made by programmers.

We applied this mutation mechanism on 130 java programs and generate 343 mutated versions with different kind of runtime violations. Among all the experimental cases, including both the original version and mutated version, there are 27 test cases without any errors; 164 test cases with some un-initialized fields. 91 test cases with different kinds of un-initialized local variables. 61 test cases with different kinds of nullification. The result of analysis by prototype is about 96% correct. We found that the missing target and false-positive results come from the following three causes: too strong hypothesis, missing the class invariants; current simple processing for iteration; lack of special mechanism to handle Java standard library.

4 Related Work and Conclusion

Control flow-based analyses are extensively used in a compiler's optimization process, which are now enhanced to do some runtime violation checking. The Java compiler is an example. Furthermore, many research works are attempting to use the data flow-based analysis. PC-lint/FlexeLint2, QA C, QA C++3, LCLint [9] and others were developed. The main drawback of those tools is that they generate a long list of warnings instead of reporting runtime violations. The advantage of our approach in comparison to flow-based approach is that our approach has reasoning capability, and can do modular checking. That is, our approach checks each class individually.

In the last decade, many researches have applied model checking to software verification, which is originally focus on hardware and protocol design [8] [9]. JavaPath Finder [10], Bandera [11], and SLAM [12] 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. Our approach does not need to construct the entire finite state model and therefore does not need huge memory or computation resources to conduct the verification process.

Past research results push researchers to think about using logical proof to reason about programs. Those effects started from Hoare triples [3], to Dijkastra's weakest precondition [1], and turns into the complete verification environments, such as

Higher Order Logic(HOL) [2], the Prototype Verification System(PVS) [4] and Extended Static Checking(ESC) system [5]. The main weakness of existing proof-based approaches is that they require the programmers to provide annotated program specification.

The main contribution of our approach to the traditional logic proof-based approach is that our approach not only uses the reasoning technology to verify source code, but also uses automated theorem prover to discover the logic information based on proper hypotheses according to specific runtime violation category. This means our approach breaks through the limitation of other proof-based approaches. Our works also show the feasibility of using formal methods to discover the implicit program specification.

Since this is still a research project undergoing, there are some limitations exists. In our future research work, we are going to make more accurate hypotheses not only in terms of violations, but also on the pattern of the source code, which will discover the weaker form of preconditions or invariants and provide more accurate result of analysis. Also in order to make our tools more practical, our future research will focus on the path reduction and analysis optimization.

References

1. Cousot, P. and Cousot R., 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation Fixpoints In *Proc. ACM SIGPLAN Conference on Programming Languages*
2. Dijkstra, E., 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Program. In *Communications of ACM. 18(8*).
3. Hoare, A., 1969. An Axiomatic Basis for Computer Programming. In *Communications of ACM, 12(10)*.
4. Kemp, D. and Goodfellow G., 1990. The Official Report, technical report. In *ACM SIGSOFT)*.
5. Leino, K. and Stata, R., 1997. *Checking Object Invariants, technical report*, Digital Equipment Corporation Research Center. Palo Alto, CA.
6. Schumann, J., 2001. *Automated Theorem Proving in Software Engineering*, Springer
7. Skevoulis, S. and Jia, X., 2000. Generic Invariant-Based Static Analysis Tool For Detection of Runtime Errors in Java Programs.
8. McMillan, K., 1993. *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer.
9. Holzmann, G., 1991. *Design and Validation of Computer Protocols*, Prentice Hall.
10. Visser, W., Havelund, K., Brat, G. and Park, S., 2000. Model cheking Programs. In *15th Conference on automated Software Engineering(ASE),* IEEE Press
11. Pasareanu, C., Dwyer, M. and Visser, W., 2001. Finding Feasible Counter-examples when Model checking Java Programs. In *Proc. of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems,* Springer-Verlag
12. Ball, T. and Rajamani, S., 2001. Automatically Validating Temporal Safety Properties of Interface. In *Proc. of SPIN 2001 Workshop on Model Checking of Software*.
13. Gosling, J., Joy, B. and Steele, G., 1996. *The Java^{tm} Language Specification*, Addison-Wesley