Runtime Checking for Program Verification Systems

Karen Zee Viktor Kuncak Martin Rinard

MIT, CSAIL
Cambridge, USA
{kkz,vkuncak,rinard}@csail.mit.edu

Abstract

One of the goals of program verification is to show that a program conforms to a specification written in a formal logic. Oftentimes, this process is hampered by errors in both the program and the specification. The time spent in identifying and eliminating these errors can even dominate the final verification effort. A runtime checker that can evaluate formal specifications can be extremely useful for quickly identifying such errors. Such a checker also enables verification approaches that combine static and dynamic program analyses. Finally, the underlying techniques are also useful for executing expressive high-level declarative languages.

This paper describes the run-time checker we are developing in the context of the Jahob verification system. One of the challenges in building a runtime checker for a program verification system is that the language of invariants and assertions is designed for simplicity of semantics and tractability of proofs, and not for run-time checking. Some of the more challenging constructs include existential and universal quantification, set comprehension, specification variables, and formulas that refer to past program states. In this paper, we describe how we handle these constructs in our runtime checker, and describe several directions for future work.

1. Introduction

This paper describes a run-time checker we are developing in the context of the Jahob verification system. The primary goals of this run-time checker are debugging specifications and the program and using run-time information in loop invariant inference.

Jahob [5, 18] is a program verification system for an imperative, sequential, memory-safe language that is a subset of Java. [1] Specifications are written as special comments within the source code, so developers can compile and run programs using standard Java interpreters and runtimes. Jahob specifications are written as formulas in higher-order logic (HOL), using the syntax of the input language to the Isabelle proof assistant [20].

Jahob specifications include declarations and definitions of specification variables (similar to model fields in JML), data structure invariants, procedure pre- and postconditions, as well as assertions. Specification variables are abstract fields defined by the programmer that can be referenced in the invariant, pre- and postconditions, and assertions. In the standard program verification usage of Jahob, the data structure invariants, pre- and postconditions, and assertions are guaranteed to hold using a combination of static analysis and theorem proving. Our runtime checker ensures that these properties hold by evaluating them dynamically.

Contributions. This abstract presents the current state of our runtime checker for Jahob. The checker evaluates a subset of higher-order logic formulas containing quantifiers, set comprehensions, integer and object expressions, sets, and relations. Among the interesting features of our checker is the evaluation of certain expressions that denote infinite sets, and the evaluation of formulas that refer to old values of fields of an unbounded number of objects.

[1] Jahob's implementation language does not support reflection, dynamic class loading, multi-threading, exceptions, packages, subclassing, or any Java 1.5 features.

2. Quantifiers and set comprehensions

While Jahob's specifications are written in HOL, for practicality purposes we restrict the runtime checker to support only first-order quantification. Even with first-order quantification, however, it is possible to write formulas that cannot be executed, as is the case when the domain of the quantifier is unbounded. Consider, for example, the formula $\forall x : T.P(x)$. Note that x refers not only to all objects of type T in the heap, but to all possible objects of type T, which would be highly impractical to compute. Therefore, in most cases, the runtime checker checks only those quantified formulas where the domain of the quantification is bounded. For integers, this means that quantification must be restricted to a range of integers. For objects, Jahob has a built-in notion of the set of allocated objects, so that quantification over all allocated objects of type T is written $\forall x : T.x : AllocatedObjects \longrightarrow P(x)$. The same applies to set comprehensions, which also need to be confined to a bounded domain in order to be evaluated. (There is an interesting case in which the runtime checker can handle even unbounded quantification, which we explain in the following section.)

Even bounded domains, however, may be large, and we would like to avoid considering all objects in the heap if at all possible. For example, in the formula $\forall x : T_x \forall y : T_y.x : AllocatedObjects \wedge y : AllocatedObjects \wedge x.next = y \longrightarrow P(x, y)$, the quantified variable y is introduced for the purposes of naming and can be easily evaluated without enumerating all elements of the heap. The runtime checker handles these cases by searching into the body of quantified formulas, through conjunctions and implications, to determine if the bound variable is defined by an equality. If so, we can evaluate the body of the formula without having to enumerate a large number of objects. While it may be possible to write the same formula without introducing a quantified variable, being able to do so may not only make a specification easier to understand, it can also make its evaluation more efficient. If the bound variable appears more than once in the body of the formula, the introduction of the quantified variable identifies a common subexpression that is essentially being lifted, so that the runtime checker need only evaluate it once.

3. Specification variables

Jahob supports two types of specification variables: standard specification variables and ghost variables. These are sometimes referred to as model fields and ghost fields, respectively, as in JML [19].

A standard specification variable is given by a formula that defines it in terms of the concrete state of the program. When the runtime checker evaluates a formula that refers to a standard specification variable, it evaluates the formula that defines the specification variable in the context of the current program state.

A ghost variable, on the other hand, is updated by the programmer using special comments in the code. They behave very much like normal variables in the program. In general, they are treated similarly by the runtime checker, though in addition to standard program types such as booleans, integers, and objects, the runtime checker also supports ghost variables of types tuple and set.

When ghost variables are updated, the right-hand side of the assignment statement consists of a formula that the runtime checker evaluates to produce the new value of the ghost variable. It then stores the resulting value in the same way as it would for the assignment of a normal program variable. This formula is a standard Jahob formula and may contain quantifiers, set comprehensions, set operations, and other constructs not typically available in Java assignment statements.

Since ghost variables of type set are allowed, it is also possible to write the following code:

```
//: private ghost specvar X :: int set;
int y = 0;

//: X := {z. z > 0};
//: assert y ~: X;
y = y + 1;
//: assert y : X;
```

The above code is an interesting case because the ghost variable x is assigned to the value of an unbounded set. The runtime checker handles this case by deferring the evaluation of x until it reaches the assert statements. It then applies formula simplifications that eliminate the set comprehension. Of course, the runtime checker uses the same simplifications when presented with a formula $y \in \{z.z > 0\}$, but the above case is an interesting example of being able to check formulas that one might not expect to be able to check.

4. The old construct

The old construct is common to most program specification languages for referring to the value of an expression in an earlier state of the program. In Jahob, an old expression refers to the value of the enclosed expression as evaluated on entry to the current procedure. Unlike the old construct in JML [19], which is syntactically restricted so that an old expression can be fully evaluated in the procedure pre-state, old expressions in Jahob are not restricted in this way. While this makes the Jahob specification language more expressive, it also makes it necessary for a runtime checker to access past program state in order to evaluate such expressions.

One simple but inefficient method of providing the checker access to past program state would be to snapshot the heap before each procedure invocation. Unfortunately, this approach is unlikely to be practical in terms of memory consumption; the memory overhead would be a product of the size of the heap and the depth of the call stack.

Instead, the runtime checker obtains access to the pre-state by means of a recovery cache (also known as a recursive cache) [14] that keeps track of the original values of modified heap locations. It is implemented as a stack that behaves as follows. On entry to a procedure, the runtime checker pushes a new, empty frame onto the stack. When a write occurs, the checker notes the memory address of the write, as well as the original value of the location before the write. Subsequent writes to the same address do not require an update to the frame. When the checker needs to evaluate an old expression, it simply looks up the necessary values in the topmost frame. If it does not find a value there, that means that the heap location was not changed, and that the current value is also the old value.

There are several features of this solution worth noting. First, it takes advantage of the fact that we need only know the state of the heap on procedure entry, and not the state of any intermediate heaps between procedure entry and the assertion or invariant to be evaluated. Also, where the state of a variable is unchanged, the old value resides in the heap, so that reads do not incur a performance penalty excepting reads of old values. Finally, one of the ideas underlying this solution is that we expect the amount of memory required to keep track of the initial writes to be small relative to the size of the heap. While there is a trade-off between memory and performance—there is now a performance penalty for each write—the overhead is greatest for initial writes, and less for subsequent writes to the same location.

5. Related Work

Run-time assertion checking has a long history [9]. Among the closest systems for run-time checking in the context of static verification system are tools based on the Java Modeling Language (JML) and the Spec# system [2].

JML [19] is a language for writing specifications of Java programs. Tools are available both for checking JML specifications at runtime and for verifying statically that a program conforms to its JML specification [6]. As such, the work on JML shares at least one of the goals of the work in this paper—that of being able to use a runtime checker to aid in the process of verifying programs with respect to their specifications. The JML compiler, jmlc [8], is the primary runtime assertion checking tool for JML. It compiles JML-annotated Java programs into bytecode that also includes instructions for checking JML invariants, pre- and post-conditions, and assertions. Other assertion tools for JML include Jass [3] and jmle [17].

One of the goals in the design of JML was to produce a specification language that was Java-like, to make it easier for software engineers to write JML specifications. It also makes JML specifications easier to execute. Jahob, on the other hand, is first and foremost a program verification system, and, as such, uses an expressive logic as its specification language. The advantage of this design is that the semantics of the specifications is clear, and the verification conditions generated by the system can easily be traced back to the relevant portions of the specification, which is very helpful in the proof process.

Spec# is another system [2] for which both runtime checking and program verification tools are available. Spec# is a superset of C# and includes a specification language. The Spec# system compiles its specifications into inline checks, which may also be verified using the Boogie verifier [1]. The Spec# specifications that we are aware of do not contain set comprehensions and transitive closure expressions.

We are not aware of any techniques used to execute such specifications in the context of programming language run-time checking systems. Techniques for checking constraints on databases [4, 12, 13,15,21,22] contain relevant techniques, but use simpler specification specification languages and are optimized for particular classes of checks.

To evaluate old expressions in our specifications, we use a recovery cache, or recursive cache, a technique from fault-tolerant computing [14]. Fault-tolerant systems use recovery caches to restore the program state to a previous state in the presence of a failure.

6. Conclusions and Future Work

The Jahob run-time checker is currently built as an interpreter and is meant for debugging and analysis purposes as opposed to the instrumentation of large programs. Among the main directions for future work are compilation of run-time checks [7, 10] to enable checking of the assertions that were not proved statically [11], and combination with a constraint solver to enable modular run-time checking [16].

References

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. 4th Intl. Sym. on Formal Methods for Components and Objects*, 2005.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[3] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass–Java with assertions. In *Proc. 1st Workshop on Runtime Verification*, volume 55 of *ENTCS*, pages 103–117, 2001.

[4] P. A. Bernstein and B. T. Blaustein. Fast methods for testing quantified relational calculus assertions. In *Proc. 1982 ACM SIGMOD intl. conf. on Management of data*, pages 39–50. ACM Press, 1982.

[5] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. Using first-order theorem provers in a data structure verification system. In *Proc. 8th Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, 2007.

[6] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003.

[7] F. Chen, M. d'Amorim, and G. Rosu. Checking and correcting behaviors of java programs at runtime with java-mop. *ENTCS*, 144(4):3–20, 2006.

[8] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, April 2003.

[9] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Soft. Eng. Notes*, 31(3):25–37, 2006.

[10] B. Demsky, C. Cadar, D. Roy, and M. C. Rinard. Efficient specification-assisted error localization. 2004.

[11] C. Flanagan. Hybrid type checking. In *Proc. 33rd Annual ACM Sym. on the Principles of Programming Languages*, pages 245–256, 2006.

[12] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Eng.*, 9(3):508–511, 1997.

[13] L. J. Henschen, W. McCune, and S. A. Naqvi. Compiling constraint-checking programs from first-order formulas. In H. Gallaire, J.-M. Nicolas, and J. Minker, editors, *Advances in Data Base Theory, Proc. of the Workshop on Logical Data Bases*, volume 2, pages 145–169, 1984.

[14] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Proc. Intl. Sym. on Operating Systems*, volume 16 of *LNCS*, pages 171–187, 1974.

[15] H. V. Jagadish and X. Qian. Integrity maintenance in object-oriented databases. In *Proc. 18th Conf. on Very Large Data Bases*, 1992.

[16] S. Khurshid and D. Marinov. TestEra: Specification-based testing of java programs using SAT. *Autom. Soft. Eng.*, 11(4):403–434, 2004.

[17] B. Krause and T. Wahls. jmle: A tool for executing JML specifications via constraint programming. In *Proc. 11th Intl. Workshop on Formal Methods for Industrial Critical Systems*, volume 4346 of *LNCS*, pages 293–296, 2007.

[18] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[19] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, December 2006.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[21] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J.-M. Nicolas, and J. Minker, editors, *Advances in Data Base Theory, Proceedings of the Workshop on Logical Data Bases*, volume 2, pages 171–209, 1984.

[22] X. Qian and G. Wiederhold. Knowledge-based integrity constraint validation. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *Proc. 12th Intl. Conf. on Very Large Data Bases*, pages 3–12, August 1986.