Dynamically Discovering Likely Interface Invariants

Christoph Csallner, Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{csallner,yannis}@cc.gatech.edu

ABSTRACT

Dynamic invariant detection is an approach that has received considerable attention in the recent research literature. A natural question arises in languages that separate the interface of a code module from its implementation: does an inferred invariant describe the interface or the implementation? Furthermore, if an implementation is allowed to refine another, as, for instance, in object-oriented method overriding, what is the relation between the inferred invariants of the overriding and the overridden method? The problem is of great practical interest. Invariants derived by real tools, like Daikon, often suffer from internal inconsistencies when overriding is taken into account, becoming unsuitable for some automated uses. We discuss the interactions between overriding and inferred invariants, and describe the implementation of an invariant inference tool that produces consistent invariants for interfaces and overridden methods.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract data types, polymorphism, and inheritance*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Pre- and post-conditions*

General Terms: Algorithms, Documentation, Languages

Keywords: Dynamic analysis, invariant detection, interfaces, method overriding

1. INTRODUCTION

Dynamic invariant detection tools like Daikon [3] and DIDUCE [4] have attracted a lot of attention in the recent research literature. Such tools attempt to monitor a large number of program executions and heuristically infer abstract logical properties of the program. Empirically, the invariant detection approach has proven effective for program understanding tasks. Nevertheless, the greatest value of program specifications is in automating program reasoning tasks. Indeed, Daikon produces specifications in several formal specification languages (e.g., in JML [5] for Java) and the resulting annotations have been used to automatically guide tools such as test case generators [9, 7].

Using inferred invariants automatically in other tools places a much heavier burden on the invariant inference engine. Treating inferred invariants, which are heuristics, as true invariants means that they need to be internally consistent. Otherwise a single contradiction is sufficient to throw off any automatic reasoning engine (be it a theorem prover, a constraint solver, a model checker, or other) that uses the invariants. In this paper, we discuss how an invariant detection tool can produce consistent invariants in a language that allows indirection in the calling of code. Object-oriented languages are good representatives, as they allow dynamically determining called code through the mechanism of method overriding. The problem has two facets:

- When a method is called on an object with a different static and dynamic type, should inferred invariants be attributed to the static type, the dynamic type, or a combination?
- How can inferred invariants be consistent under the rule of *behavioral subtyping*, which states that the overriding method should keep or weaken the precondition and keep or strengthen the postcondition of each method it overrides.

We discuss these issues in the context of Java, the JML specification language, and the Daikon invariant inference tool. Similar observations apply to different contexts. We describe a solution and our in-progress implementation of a dynamic invariant inference tool that supports it.

2. BACKGROUND AND MOTIVATING EXAMPLES

We begin with the necessary Daikon and JML background to give motivating examples of the problem.

2.1 Daikon, JML, Behavioral Subtyping

Daikon [3, 8] tracks a program's variables during execution and generalizes their observed behavior to invariants—preconditions, postconditions, and class invariants. Daikon instruments the program, executes it (for example, on an existing test suite or during production use), and analyzes the produced execution traces. At each method entry and exit, Daikon instantiates some three dozen invariant templates, including unary, binary, and ternary relations over scalars, and relations over arrays (relations include linear equations, orderings, implication, and disjunction). For each invariant template, Daikon tries several combinations of method parameters, method results, and object state. For example, it might propose that some method `m` never returns null. It

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

later ignores those invariants that are refuted by an execution trace—for example, it might process a situation where m returned null and it will therefore ignore the above invariant. So Daikon summarizes the behavior observed in the execution traces as invariants and generalizes it by proposing that the invariants might hold in all other executions as well. Daikon can annotate the testee's source code with the inferred invariants as preconditions, postconditions, and class invariants in the JML [5] specification language for Java.

JML enforces the principle of *behavioral inheritance* or *behavioral subtyping* [5] for overriding methods. Informally, behavioral subtyping is the requirement that the overriding method should be usable everywhere the method it overrides can be used. This is a common concept, employed also in program analyzers (e.g., ESC/Java2 [2]) and design methodologies (e.g., "subcontracting" in Design by Contract [6]). To see behavioral subtyping more formally, consider the following Java code with JML annotations:

```
public class Super {
  //@ requires P;
  //@ ensures Q;
  public void m() {...} }
```

JML requires that a subclass's preconditions and postconditions be specified with the `also` keyword, to denote behavioral subtyping:

```
public class C extends Super {
  //@ also
  //@ requires R;
  //@ ensures S;
  public void m() {...} }
```

R and S are *not* the precondition and postcondition of method C.m, however. Instead, JML derives the preconditions and postconditions from the `also` clauses and the behavior of the overridden method Super.m:

- C.m's precondition is `P | R` (read "P or R")
- C.m's postcondition is `(P ==> Q) & (R ==> S)` (read "if P holds as a precondition, Q holds as a postcondition and if R holds as a precondition, S holds as a postcondition").

This is exactly the formal embodiment of behavioral subtyping: the precondition of the subtype method allows all the preconditions of the methods it overrides, plus possibly some more. The postcondition of the subtype method is at least that of the overridden method if the precondition falls inside the original domain, and may also have more constraints.

2.2 The Problem with Overriding

With the above background, we can now see the two issues arising in the interaction of invariant inferencing tools and overriding.

The first issue is whether a precondition or postcondition is really a property of the static or the dynamic type of an object. Daikon associates any method execution with the executed method body, not with the method definition of the call's static receiver. It then infers invariants from the execution trace, maintaining the association with the executed method. According to this behavior, Daikon never infers pre- or postconditions of methods defined by a Java interface, since interface methods do not have a body. Yet, this behavior is counter-intuitive: even though postconditions are a property of the called code, preconditions are established by the calling environment. When these are inferred by actual program behavior, they should also be associated with the type known to the caller, regardless of the actual type of an object.

In the following example, the Client class was written against interface I. Method Client.foo calls I.m, so foo has to honor all preconditions of I.m. (These might be specified informally in the Javadoc comments of I.m or elsewhere.) The conditions that hold when method m gets called reflect the preconditions of the abstract method I.m, and not just those of the called method Impl.m.

```
public interface I {
  public void m(int arg); }

class Client {
  void foo(I i) {  //called with i = new Impl()
    i.m(...); } }

public class Impl implements I {
  public void m(int arg) {...} }
```

The second issue with invariant inferencing and overriding is thornier. Since invariant inference is heuristic, it is easy to derive invariants that do not respect behavioral subtyping and, thus, lead to a contradiction. Consider a method m defined in a class Super and overridden in a subclass C. Assume a scenario where, under the observed program behavior, Super.m is consistently called with an input value of 1 and always returns (in the observed executions) the output value 1. It is reasonable to infer precondition `i == 1` and postcondition `\result == 1` for Super.m. At the same time, if C.m is also consistently called with input value 1 and always returns (in the observed executions) the output value 0, then it is reasonable to infer precondition `i == 1` and postcondition `\result == 0` for C.m. Daikon just outputs the invariants for both methods, with the `also` clause used for the invariants of C.m, as dictated by JML for overriding methods:

```
public class Super {
  //@ requires i == 1;
  //@ ensures \result == 1;
  public int m(int i) {...} }

public class C extends Super {
  //@ also
  //@ requires i == 1;
  //@ ensures \result == 0;
  public int m(int i) {...} }
```

Then, according to the JML rules discussed earlier, the complete invariants for C.m become:

- Precondition: `i == 1`
- Postcondition:
  `((i==1) ==> (\result==1)) &`
  `((i==1) ==> (\result==0))`.
  We can simplify this to:
  `((i==1) ==> (\result==1) & (\result==0))`,
  which is equivalent to: `i != 1`.

The derived precondition means that calling `C.m(1)` is legal. It is impossible for the method to reach its postcondition, though, since the method cannot change the state (`i == 1`) that existed before its own execution. (Any method parameter appearing in a postcondition is evaluated to its value before method execution—it is implicitly enclosed by `\old`). So the method cannot terminate normally without violating its postcondition. But it cannot go into an infinite loop or terminate abnormally by throwing an exception either, since such behavior is ruled out in JML when left unspecified. Thus, every possible implementation of method `C.m` (including the actual implementation observed by Daikon) violates the derived specification.

This contradictory postcondition is very undesirable for any automatic use of the derived specifications. The problem is that *the behavior of the overriding method,* `C.m`*, indirectly reveals that there is more behavior of the overridden method,* `Super.m`*, than seen during the execution of the test suite.*[1] Nevertheless, there is no easy way to take this into account during the inference of the invariants for method `Super.m`. It is tempting to think that there may be a different set of conditions that can be output for `C.m` so that no contradiction occurs. While we could explicitly manipulate the `C.m` invariants to narrow the precondition (in this case to `false`) to address the contradiction, this would also render the invariants of `C.m` useless. The problem is fundamentally with the invariants of `Super.m` and not `C.m`.

3. SOLUTION DESIGN

To solve both problems described earlier, we designed a general algorithm that invariant detection tools can follow. The algorithm is oblivious to the actual strategy of the tool for deriving invariants from executions, and instead concentrates on what method observes which execution (i.e., which input and output values). The algorithm can be described informally as follows: values at input are used for computing the precondition of the method executed (dynamic receiver) and all methods it overrides up to and including the static receiver. Values at output are used to compute the postcondition of the method executed and all methods it overrides as long as the values satisfy the methods' preconditions.

Figure 1 illustrates the algorithm, which has the following two phases:

- Phase A: the test suite is run and values at the beginning of each call are registered for the dynamic receiver of a method call and for all methods it overrides up to and including the static receiver. Preconditions are inferred from these values, in the same way as they would be otherwise. No postconditions are inferred. The phase completes with all preconditions computed.

- Phase B: the test suite is run again (or a trace is replayed) and for a call to method `C.m` with inputs `i1..iN` we find all methods (including `C.m` itself) that `C.m` overrides. For each such method, `S.m`, if the inputs `i1..iN` satisfy the inferred precondition of `S.m` then the execution of the method is used in the computation of the postcondition of `S.m`.

[1]The implicit assumption for every dynamic invariant detection tool is that the derived invariants have to be consistent with the observed behavior. So a future execution of the observed behavior should pass any invariant checks compiled from the derived invariants.

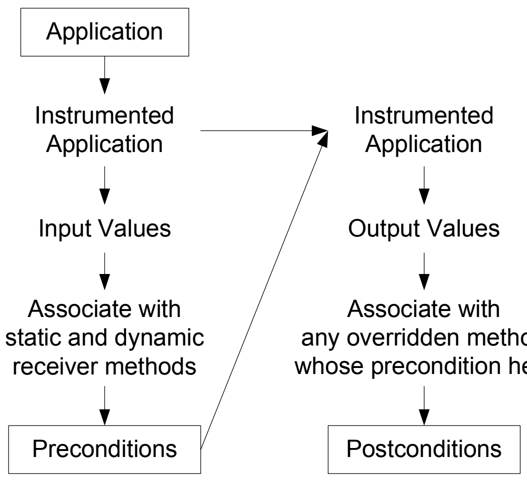

Figure 1: Phase A (left) and Phase B (right).

This approach solves both problems identified earlier. First, preconditions are computed for both the static and the dynamic receiver of a method call. Second, an execution of an overriding method is also used to compute postconditions for all its overridden methods, as long as the inputs do not fall outside the domain of the overridden method. Since that domain is fixed (from Phase A), we are guaranteed that no contradictory postconditions can be computed (because the overriding and overridden methods have seen the same behavior for all inputs in their common domain).

Note that there are several other ways to remove the symptoms of the second problem (i.e., the derivation of a contradiction). Generally, we can strengthen (i.e., narrow) the inferred precondition of the overriding method or weaken its postcondition until the contradiction disappears. Nevertheless, all such approaches result in artificial overapproximations. In contrast, our above algorithm solves the problem by ensuring that we take into account all relevant behavior for every method when computing its pre- and postcondition.

4. SOLUTION IMPLEMENTATION

We are in the process of implementing a variant of Daikon using the above algorithm to guide the invariant inference logic. There are several implementation complications in adopting our approach. First, we need to efficiently carry information about the static type used to call a method during method execution. Then, we need to compile inferred JML preconditions into actual runtime checks that we will perform during Phase B of our algorithm to determine which methods' postconditions may be affected by a given execution.

4.1 Keeping Track of Static Receivers

For the first issue, consider a method `Impl.m` dynamically dispatched through an interface `I.m`. The problem is that the executed method (`Impl.m`) does not have direct access to the static receiver type (`I`) against which the method was called (Figure 2).

We could rewrite the call site to observe the entry values and possibly update the static receiver's preconditions.

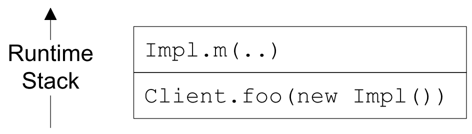

Figure 2: The static receiver type `I` is not readily available during the execution of `Impl`.

Nevertheless, this is awkward, since a similar task also needs to be performed inside the body of the dynamic receiver. Instead, we want to pass the static receiver type information to the dynamic receiver body to use its existing invariant inference routines. To avoid synchronization problems with centralized data stores we transform methods to pass the static receiver type information with the method call (Figure 3). That is, a call to method `m` in class `Impl` will be transformed to calling the jump-through method `m___I` for all its calls through a reference with static type `I`:

```
public interface I {
  public void m(int arg);
  public void m___I(int arg); }

class Client {
  void foo(I i) {  //called with i = new Impl()
    i.m___I(...); } }

public class Impl implements I {
  public void m___I(int arg) {
    trace.add(I);
    return m(arg);
  }
  public void m___Impl(int arg) {
    return m(arg);
  }
  public void m(int arg) {...} }
```

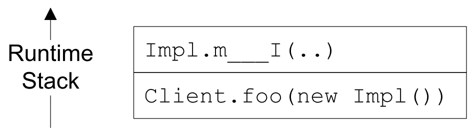

Figure 3: The static receiver type `I` is encoded in the method name `m___I`.

We use BCEL [1] for the program transformation at the bytecode level.

4.2 Checking Invariants During Execution

After Phase A we add the derived preconditions as special checks to the application. At the beginning of each method body we add instructions that check if the precondition of any overridden method holds as well. We associate the traced values with every so determined method.

Up front we determine all method definitions that are overridden by a given method `D.m`. Note that `m` may override methods in more than one direct super-types (e.g., in interfaces) and in transitive super-types. If `m` has a method body (is not abstract), we compile the precondition derived for every overridden method into a separate runtime check and add it to the beginning of `m`. If a runtime check succeeds then the current invocation also satisfies the precondition of the overridden method and we associate the invocation with this method definition. In the following example, an invocation of method `D.m` may contribute to the postconditions of `C.m`, `B.m`, and `A.m`, in addition to `D.m`.

```
public interface A {
  public void m(int i); }

public interface B extends A {
  public void m(int i); }

public class C {
  public void m(int i) {...} }

public class D extends C implements B {
  public void m(int i) {
    if A.m.pre(i) {trace.add(A);}
    if B.m.pre(i) {trace.add(B);}
    if C.m.pre(i) {trace.add(C);} ... } }
```

5. CONCLUSIONS

We presented an algorithm for dynamic invariant detection that supports interfaces and method overriding. We have outlined an implementation in the context of the Daikon dynamic invariant detector. Future work includes an empirical evaluation of the algorithm.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. CCR-0220248 and CCR-0238289.

6. REFERENCES

[1] Apache Software Foundation. Bytecode engineering library (BCEL). http://jakarta.apache.org/bcel/, Apr. 2003. Accessed Feb. 2006.

[2] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2. Technical Report NIII-R0413, Nijmegen Institute for Computing and Information Science, May 2004.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[4] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, May 2002.

[5] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR98-06y, Department of Computer Science, Iowa State University, June 1998.

[6] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 1997.

[7] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. 19th European Conference on Object-Oriented Programming*, pages 504–527, July 2005.

[8] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, Nov. 2004.

[9] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. 18th Annual International Conference on Automated Software Engineering (ASE 2003)*, pages 40–48, Oct. 2003.