Dynamic Invariant Detection for Program Understanding and Reliability
MIT2001-01

Michael Ernst

This research will exploit dynamic detection of program invariants over existing software to prevent it from degrading when modified, to improve its reliability, to detect flaws, and to enhance programmer understanding. Three specific projects are proposed: static verification of dynamically detected invariants, test suite quality measured by dynamically detected invariants, and dynamic detection of temporal invariants. These projects aim to both advance the state of the art and to assist NTT in evaluating the technology and putting it to productive use.

Background: Dynamic detection of program invariants

Program invariants, including dynamically detected ones, are also helpful in building new programs.

Program invariants are properties that are true at a particular program point, such as are found in formal specifications and assert statements. Invariants provide valuable documentation, can test program operation and expose bugs or special cases, can double-check textual documentation, can assist in test case generation or validation, and can bootstrap or direct proofs.

Most programs lack explicitly-stated invariants, depriving programmers of these advantages. Dynamic invariant detection recovers these program invariants from program executions. The user runs the target program over a test suite to create the traces, and an invariant detector determines which properties and relationships hold over both explicit variables and other expressions. Properties that hold over the traces and also satisfy other tests, such as being statistically justified, not being over unrelated variables, and not being implied by other reported invariants, are reported as likely invariants. The results can be communicated back to a human programmer or fed to another tool. Dynamic invariant detection is implemented in a tool called Daikon.

Like other dynamic techniques such as testing, the quality of the output depends in part on the comprehensiveness of the test suite. If the test suite is inadequate, then the output indicates how, permitting its improvement. Dynamic analysis complements static techniques, which can be made sound but for which certain program constructs remain beyond the state of the art.

Preliminary research on dynamic detection of program invariants has been very promising. Formal specifications were recovered from formally specified programs, and in non-formally-specified programs, the dynamically detected invariants assisted programmers in a software maintenance task. In both cases, bugs were also found. Dynamic invariant detection tends to run quickly and produce output of modest size, for the relatively small programs examined to date. Test suites found in practice tend to be adequate for dynamic invariant detection.

Dynamically detected invariants can identify opportunities for refactoring; refactoring restructures software to improve desirable qualities such as performance, maintainability, or flexibility. Additionally, dynamically detected invariants can be statically proved. Combining the two techniques thus eliminates a potential objection about the unsoundness of invariant detection, and it eliminates a potential objection about the difficulty of annotating programs for statically verification.

The proposed research will both build on previous research for dynamic invariant detection and apply it to specific areas of interest to NTT. For instance, the techniques could be applied to cellular phone programs, to network code for digital protocols, or similar application areas. We expect that case studies of NTT software will identifying weaknesses in and improve both the NTT software and the Daikon invariant detector toolset.

We propose three specific projects, with the expectation that our counterparts at NTT will select the one(s) of most interest to them.

Proposed Project 1:

Static verification of dynamically detected invariants

Dynamic detection and static verification are complementary techniques for manipulating program invariants. Dynamic detection can propose likely invariants based on program executions, but the resulting properties are not guaranteed to be true over all possible executions. Static verification can check that properties are always true, but it can be difficult and tedious for people or programs to select a goal and to annotate programs for input to a static checker. Combining these techniques overcomes the weaknesses of each: dynamically detected invariants can annotate a program or provide goals for static verification, and static verification can confirm properties proposed by a dynamic tool.

We will integrate a tool for dynamically detecting likely program invariants, Daikon, with a tools for statically verifying program properties, such as ESC (for Java), LCLint (for C), or the Larch Prover (for the IOA programming language). These tools take as input a program annotated with preconditions, postconditions, and other assertions, and reports which annotations cannot be statically verified and also warns of potential runtime errors, such as null dereferences and out-of-bounds array indices. We will determine whether this integration is feasible (preliminary experiments suggest it is) and the limits of its usefulness.

Proposed Project 2:

Test suite quality and dynamically detected invariants

The invariants reported by a dynamic invariant detector are sometimes functional invariants about particular procedures or modules which are true regardless of how those program components are used, and sometimes the outputs are usage properties which reflect how the component happened to be used during the test runs being examined. Both sorts of invariants are useful, but for different purposes. (The project to statically verify dynamically detected invariants would be one way of determining which variety a reported invariant is.)

Usage properties can indicate valuable information about the program's context, such as how modules are used, and about the test suite. For instance, knowing that a test suite only calls a procedure with a positive argument can indicate that it ought to be tested with zero and negative arguments; dynamic invariant detection reveals value coverage, which is a valuable adjunct to code coverage when evaluating test suites. (Alternately, if the procedure is necessarily used with a positive argument, that information can be documented and the procedure can be optimized based on knowledge of how it is used.)

To date, test suites used for finding bugs have been useful for invariant detection. We propose to perform more experiments to understand the relationship between standard test suite quality criteria, such as coverage, and the quality of the invariants that are detected when the program is executed. We will also perform case studies to determine how useful dynamically detected invariants are to testers. Invariant detection can give feedback about the quality of the test suite, such as properties that are always (or never) true for the test suite. This could provide a valuable tool for software testing.

Proposed Project 3:

Dynamic detection of temporal invariants

Currently, the prototype invariant detector reports only invariants that are true at one or more particular points in time (for instance, upon entry to all public routines in a module). Temporal invariants, by contrast, relate properties or facts at multiple points in time. As a simple example, consider a traffic signal. A static or data invariant might state, "no two directions show green simultaneously". A temporal invariant might state, "if a car arrives at the intersection, then the light will eventually turn green.

Temporal invariants are typically written in a temporal logic such as LTL. They relate events with operators such as "eventually" or always.

We propose to extend our prototype to detect temporal invariants as well as static data invariants. The basic technique is the same: examine many data, extrapolate a pattern by generalizing those data, and then check that pattern over all the remaining data. Challenges to this research include the much larger amount of data, and the much longer time before a proposed invariant can be falsified or verified. We will determine to what extent current techniques will work for temporal invariants and discover new techniques to improve the results, if necessary.