Invariants and State in Testing and Formal Methods

Dick Hamlet*
Portland State University
Portland, OR, USA
hamlet@cs.pdx.edu

ABSTRACT

Logical formulas called invariants are a staple of formal methods for program analysis. Persistent-state variables appear in these formulas playing their proper intuitive role, which is somewhere between inputs and internal variables. In software testing theory, on the contrary, state is not usually accorded explicit treatment. Comparing the viewpoints of formal methods and testing theory suggests new roles that formal methods can play in testing. This examination is motivated by recent systems such as Daikon, which use both tests and invariants.

1. PROVING AND TESTING

It is a truism that program proving ('formal methods') and software testing are at opposite ends of a spectrum of software analysis techniques. One is static, the other dynamic. One can be sound and complete, the other cannot be. One has the reputation that it requires a doctorate to use, the other is routinely employed by untrained technicians. Sometimes the methods are said to complement each other, but this usually means only that they are unrelated, not that one aids the other. Apart from one promising early paper [3] that used test results in proofs, there have been few suggestions about how one technique can benefit the other.

This paper utilizes the disparate viewpoints of testing and formal methods to discuss and illuminate two concepts of conventional imperative-language programming: persistent state and program invariants. These ideas are connected in that persistent-state variables appear in invariants, where they play a role somewhere between input variables and internal program variables.

It is helpful to describe testing and proving theory in the simplest terms, to focus attention on fundamental issues rather than the extensive details of practice. To this end a program is taken as having an input domain D, an output range R, and a state space H. Programs are imagined to read from D, write to R, and read/write from/to H. D and R can be thought of as the "standard input" and "standard output" of UNIX systems. H can be thought of as a permanent disk file whose contents are carried from one execution to another. If we ignore the details of what occurs while a program is in execution, the semantics of a program is a mapping $D \times H \longrightarrow H \times R$. The reader who would like to think in terms of program variables and their data types can imagine that the sets D, R, H have some particular type (say "integer") and each corresponds to a single variable. Then the semantics of a program maps a pair of integers into another pair.

The state space H is an anomalous entity in the theory of programs. On the one hand, it acts like a kind of input space (or independent variable) in that program behavior depends on the state value when execution begins. But unlike the input domain D, values in H do not really vary independently—the program itself creates them. Understanding this anomaly is crucial for the theory of testing programs with persistent state. The tester can supply state values and the program will use them, but to do so can be misleading. An arbitrary state value not created by the program may have no significance. The anomalous nature of state also arises in formal specification. It seems very difficult to describe what a program is required to do without describing its use of persistent data; yet, such a description seems too prescriptive. Is there not a better specification using only input-output values? Should state not be left to the program designer as other data-structure decisions often are?

Within this simplified context, let us describe the testing and proof viewpoints.

1.1 Testing Theory

Testing theory began in a paper by Goodenough and Gerhart [4] by treating a program under test as a black box with pure functional behavior mapping $D \longrightarrow R$. Test points are samples from D. A specification function F with the same domain and range as the program function is the arbiter of what the program should do. A test input $x \in D$ fails if the program output at x disagrees with $F(x)$.

It was not immediately recognized that the quality of a test involves the *operational profile* of the program being tested—a probability distribution on D describing its use. To have significance, samples (test points) must be selected from this distribution[1]. It is sampling from an arbitrary operational profile that gives testing its unique character.

To account for persistent state in testing theory, D must be replaced by the set of all its sequences: $D^\infty = \cup_{k=1}^{\infty} D^k$, and the program function and specification (still pure functions) map sequences from D (that is, individual values from D^∞) to outputs. The operational profile for a program with persistent state must thus give relative weights to values in D^∞, sequences of inputs.

*Supported by NSF ITR grant CCR-0112654 and by an E.T.S. Walton grant from Science Foundation Ireland. Neither institution is in any way responsible for the statements made in this paper.

[1]Goodenough and Gerhart, and other theorists who followed them, instead concentrated on defining test collections that are sufficient to prove the program correct, with the unsurprising result that such collections exist but cannot be effectively found.

Any such input sequence gives rise to a sequence of state values from H, but the tester need not be concerned with these since the program creates them. Unfortunately, in practice, input sequences may be difficult to generate and sample, and a profile for sequences may be unknown, forcing the tester to resort to sampling H explicitly. (This will require that the specification function be given in terms of H as well[2].) Using H in this way is technically incorrect, because values chosen may be ones that never arise from any input sequence. In Section 2.2 the question of sampling H will be considered in detail.

1.2 Floyd/Hoare Proof Theory

Proof methods that originated with Floyd [2] and were developed by Hoare [7] have always taken persistent state into account. In the simple context considered here, a program would have only three variables, each taking values from D, H, and R, which we will call the input-, state-, and output variable respectively; we will use the usual notation of a corresponding lower-case variable name. Thus "d" is the input variable and/or its value. The first-order assertions that express what should be true at a given point in a program are written in terms of the values of program variables there; the state variable h is treated like any other. A precondition in first-order logic is written in terms of d and h values before the code is entered. A postcondition is written in terms of $r, d,$ and h (values after execution) and d' and h' (before). A program is correct if its precondition implies its postcondition.

The special role played by state enters Floyd/Hoare theory in the form of *invariants*. Invariants are also first-order formulas, which if assumed to hold before code is executed must be proved to hold afterward. A proof obligation for a loop invariant, for example, is that if it holds before the loop body is executed, then it must hold after body execution. In principle, proofs of correctness do not require invariants. But an invariant may be essential to the proof, particularly for a mechanical theorem prover. Intuitively, an invariant restricts variable values and so eliminates cases that cannot occur yet would have to be considered in a proof without the invariant. As a simple example, an invariant establishing for a program that state h has a single fixed value h_0 simplifies any assertion about that program from universal quantification over h to the single case of h_0.

The name 'invariant' can introduce confusion because the logical formulae that are to be proved invariant are sometimes given that name *a priori*, giving rise to peculiar statements like, "The invariant is not preserved." (I.e., it isn't invariant!) The nomenclature problem is particularly noticeable in recent work where tests are used in conjunction with logical formulas. In Daikon [1], for example, a particular formula Q is checked against test data and some test point may "falsify the invariant Q." (Such naming problems are not new, and "equation" itself, applied to an unsatisfiable formula like $x = x + 1$ that happens to contain an "=" sign, is an example.) Systems like Daikon bring together the concepts of testing and formal methods, and they are a major motivation for this paper.

Model-checking technology has brought another formal view of state to further complicate the picture. Formal properties of a state-based model are expressed in a temporal logic, in which 'time' represents the passage of a program from state to state as it executes. By demarking the "state" variables and particular points where executions begin and end, a temporal formula can be made to describe execution sequences. The difficulty of describing sequences formally is reflected in temporal-logic usage by the scarcity of complete specifications: usually temporal formulas describe only narrow special properties. This paper largely ignores temporal-logic formalism, concentrating on the more fundamental Floyd/Hoare theory.

[2]It illuminates the intuitive deficiency of specifications that explicitly prescribe state to consider the case of a program that gives correct output values for all input sequences, but fails to give state values that meet the specification. The intuition is that the state was over-specified, and the implementor properly ignored it.

2. TESTING WITH FORMAL METHODS

The difference in viewpoint between testing theory and formal methods has limited the application of formalism in testing, but systems like TestEra [8] and Daikon [1] incorporate logical formulas. These systems have been presented from the formal viewpoint that testing approximates formal proof. Here we consider instead the testing viewpoint.

The archetype situation to consider is a post-condition formula L that is written using the original value of an input variable d', and a persistent-state variable (original value h' and final value h). If L is taken as a specification that code must satisfy, then a correctness proof must demonstrate that L is true following an arbitrary execution of the code. (For simplicity, assume that there is no pre-condition other than 'true'.) Execution of the code establishes two relationships: (1) among the input and output variables, and (2) between the original and final values of state variables. These relationships are the substance of a correctness proof. In testing, however, the program meaning takes care of itself. For any given values of the input and the state, the code can be executed and L checked.

2.1 Invariants and Post-conditions in Proofs

A way to look at formal-methods ideas from the testing perspective is to imagine that a post-condition formula L is itself being tested. If there were no persistent-state variable in L, then a proof of correctness would be obtained by demonstrating that L, universally quantified over its input variable d', holds at the end of execution. The testing approximation to this would be to sample D, run the program, and evaluate L.

With persistent state things are more complicated. It would prove correctness to demonstrate L universally quantified over d' and the original value of the persistent-state variable h'. The corresponding testing approximation would be to sample D and H, run the program, and check L. But quantification over H yields too strong a proof obligation because it ignores the relationship the program establishes between h' and h. If a state value cannot be reached no matter what sequence of inputs from D is supplied, the proof need not consider it. It is precisely the role of an invariant to eliminate such spurious states. Suppose then that in addition to the post-condition L we have a formula I (in variables d and h) expressing a necessary property of persistent-state data. If I is indeed an invariant, that is, if assuming it as precondition it can be demonstrated to hold as postcondition, then instead of demonstrating L universally quantified over state variables, it is enough to demonstrate $I \Rightarrow L$. There is a 'strongest possible invariant' that implies all others, but it may not be necessary to find and use it; the proof might go through with a weaker I.

2.2 Invariants in Testing

Invariants and post-conditions name the state variable h, so if testing is to be thought of as probing these formulas, the tester will be making choices from H as well as from the input domain D.

First, I itself can be tested, by sampling $D \times H$. For one test point $(d, h) \in D \times H$, if I holds, the program is executed and I must remain true. If I does not hold for (d, h), no execution is needed. Testing an invariant is quite unlike testing a postcondition because initial failure of an invariant demonstrates nothing except the invariant's strength. Furthermore, checking invariance—that is, that the formula is *preserved* over a test execution—is not usual in testing. When invariance fails, it detects a special kind of program failure in which an assumption about data is violated. Testing cannot of course determine the quality of a potential invariant—one never knows if a given formula would continue invariant on test points untried or whether some stronger formula might be invariant.

Second, once I holds for an input, the post-condition L can be checked after execution.

Thus a choice of invariant can drive the testing process. Insofar as I eliminates testing of cases that cannot in fact occur, it concentrates testing of L on real cases. If, for an extreme example, the state space H is the integer interval $[1, 100]$ and an invariant is $h \in \{1, 2\}$, then only 2% of test points chosen fairly from H have any meaning and the invariant filters out the irrelevant 98%. It should be much easier to deduce weak invariants from an informal specification than to come up with the stronger post-conditions.

There is an conceptual parallel between an invariant and an operational profile. A profile for a program with state is a probability density over sequences of inputs. In these sequences, any actual data invariant I holds between each execution and the next. When instead choices for input and state values are made without the profile, in effect the tests constitute probes into the sequence space. So long as the state h in a test-point choice (d, h) satisfies the strongest possible I, the test point might occur in some sequence. The invariant seems to be acting like a kind of formal profile to weight test selection, but it lacks the arbitrary character of an operational profile. A test profile does not specify what a program should or does do; it describes only how some human user intends to use the program.

Given the difficulty of exploring the sequence space, the tester is likely to directly sample the state space. Let f be an operational profile, that is, a probability density function on D^∞ giving the chance that any particular sequence will occur in use. f induces a projection f_H into the state space. Intuitively, $f_H(h)$ is the frequency with which $h \in H$ occurs in operational sequences, which may be obtained by counting the occurrences of h in a sequence $d_s \in D^\infty$, weighting by $f(d_s)$, then normalizing over all sequences. Using this projection, the tester can make weighted independent choices for state values, and know that the values chosen occur often in operational sequences. A great deal of information about f is lost in the projection f_H, however.

In the common case that a sequence profile f is not available, making a guess for f_H allows a test engineer to work systematically. In particular, a data invariant I can be used to define a state profile like f_H as a uniform sampling of all states for which I holds. A strong invariant will avoid more impossible states, but can be arbitrarily different from any actual operational profile.

2.3 'Test-based' Logical Formulas

Any logical formula R concerning a program's behavior—an invariant, post-condition, etc.—has a dual character. On the one hand, R may describe what is true of the program in all possible cases; this could be called a 'proof-based' view of R. But a contrasting 'test-based' view of R comes from an operational profile f: when test sequences are selected according to f, there is a high probability that R will be true[3]. Test quality should be judged by how R is covered. The proof-based view would be that a good test makes a wide-ranging probe into R—the test approximates the universal quantification of proof. The test-based view is quite different: a good test is one that hits more of the states an operational profile singles out. The accurate test-based view is that these states must occur in order, in sequences given by a profile f defined on sequences. The more practical view is that there is a projected or guessed state-space profile f_H, and good tests agree with the f_H weighting. When f_H itself is defined by an invariant I arising from the specification as in Section 2.2, I itself might be called a 'test invariant' or a 'proof invariant'.

[3]Ernst calls these 'proof-true' and 'testing-true' formulas.

The proof-based and test-based views may be very different because a profile defining the test-based view is arbitrary. It may choose to neglect most of the space over which R is defined. Our first intuition (clearly 'proof-based!) is that a test that does not explore much of R is a poor one. But someone holding the test-based view would counter that if cases do not occur in use, they are irrelevant.

2.4 An Invariant Example

Consider a program that manages cooperative work on a collection of files, sometimes called a source-code control system (SCCS). Its inputs are commands to check-out (CO) and check-in (CI) files, and part of its persistent state keeps track of file status. In the algorithm used in the CVS SCCS there is no file locking, so the state records versions and lists of their users along with possible conflicts created by multiple updates of the same version. A typical state might include: "file F checked-out as version 7 by user X then checked-in as version 8; file F checked-out as version 7 by user Y, no conflicts with version 8". In this state, user Y should be allowed to check-in F, from which the SCCS would create version 9 merging user Y's changes with those made by user X in version 8. If we ignore race conditions, use of this SCCS is a sequence of CO and CI inputs from multiple users for multiple files. In possible sequences, some state values should never appear. For example: no checked-out version number of a file should be larger than the last checked-in version; there should not be a file checked-in by a user who never checked it out; etc. A data invariant can capture such dependencies and drive testing of the SCCS as described in Section 2.2.

A sequence profile f for the SCCS captures the frequencies of usage. For example, in one environment most sequences might involve a single file, many users, and no conflicts. From this information a tester can restrict state sampling to common cases that are much narrower than those induced from even the best data invariant. The projection f_H of a given sequence profile f goes beyond intuitive state sampling, since it supplies relative weights for how often states appear. But even f_H falls short of testing using f—only f itself captures information like the way in which users' requests are distributed in a sequence.

2.5 Test-case-driven 'Specifications'

Systems that use tests to generate specification-like entities motivated this examination of state and invariants. We now examine Daikon's 'invariants'[4]. Daikon uses test data to probe behavior of a program and tries to produce a description of what is observed in proof-based terms. The formulas generated hold for the data that has been tried, but might be falsified by additional tests. The question considered above was: "How can we test using given formulas?" In Daikon it is turned around into: "When we have tested, what formulas can we say are likely to hold?"

In Daikon, terminology is a problem because its 'invariants' may not only be falsified, but are really candidates for post-conditions or pre-conditions within the program. Even the term *assertion* is too loaded for Daikon. Assertion use in testing and debugging

[4]Henkel [6] has devised a system with a similar purpose, but using the formalism of algebraic equations. The different formalism matches somewhat differently with testing theory, but space limitations preclude its discussion.

of imperative programs is important and well established [9], but Daikon's use of logical formulas is different. Normally, an assertion arises from a given external specification, and is placed in an implementation to check that the specification is being observed. Should the assertion fail for some test execution, it flags an inconsistency between program and specification. Daikon's formulas, on the other hand, are generated from a fixed list of logical patterns; a generated formula is discarded if it is falsified by test data supplied to the program. In this discussion we will stick with the awkward term, 'Daikon logical formula,'[5] DLF for short.

When a collection of test cases has been run and a DLF holds for them all, then Daikon's usage parallels the normal use of assertions, with the crucial difference that the DLF has no specification source—it arises from Daikon's list and the program alone. A DLF that is bolstered by test data, whatever its source, is not of course proved, because it may fail for additional tests that have not been tried. But Daikon calls its so-far-unfalsified DLFs 'invariants.'

As implemented, Daikon leans toward proof-based DLFs. Some formulas are not reported despite agreement with all the test data, because few test cases justify the formula. This happens in two ways: (1) The DLF is seldom encountered under test; and (2) The DLF is frequently encountered, but not with a wide range of values for its variables. If a test-based criterion were being used, the decision (2) would be reversed: repeatedly encountering the same values would justify the DLF as a test-based 'invariant.' Daikon does not generate test data, and insofar as the data that a user supplies is drawn from a profile of some kind, it emphasizes the test-based validity of making the opposite choice for (2).

Daikon does not deal in formulas that play the role of data invariants, precisely because these cannot be generated and then winnowed by falsification under test. Daikon generates pre-condition DLFs observed to be true before test execution begins, but these are not true invariants. Data invariants are inherently prescriptive, arising only outside a program. So long as the tester avoids any direct sampling of persistent state, the preconditions Daikon derives are data invariants the program has observed; but if state is sampled directly, only an outside assertion can filter out cases that are not meant to occur, cases which if used might falsify post-conditions meant to hold. A strong post-condition may be true for executions satisfying a given data invariant, but fail when the invariant fails. To add specification-derived data invariants to Daikon might significantly improve the test-based character of its generated post-conditions. Although such a change would be large in principle, its implementation probably would not be difficult.

For example, a program might use a persistent array and slice bounds within it to store a more restricted structure. A data invariant can describe the meaningful relationship between the bounds and the properties of array elements between them. This will preclude test data in which the bounds and properties do not match, so that a post-condition can specify that the properties are universal. Should the program fail to keep the array in proper format for some input, the data invariant will hold prior to that execution and fail after it.

Daikon is being used to investigate a number of testing issues, notably the relationship between DLFs and traditional structural coverage criteria (see [10] for a bibliography). Gupta and Hiedepriem [5] find test data to 'cover' a DLF obtained from structural-coverage tests, using def-use dependencies of the variables in the DLF. None of the recent work considers persistent state or true specification-based invariants.

3. SUMMARY

Testing theory has not explicitly included the ideas of persistent state and data invariants limiting values of state. Instead the theory has considered sequences of inputs and only implicitly the state values that arise in these sequences. In practice, however, a tester usually sample states directly. Unfortunately, arbitrary choices of state can create spurious executions that are not part of any input sequence. The machinery of Floyd/Hoare-based formal methods can be used to drive better state-space sampling. In testing using a specification-based invariant I: (1) I eliminates spurious tests; (2) If I is not invariant for the program, it signals a new kind of failure to meet specifications, one that is not currently tested.

Invariants to guide testing should be easier to derive from informal specifications than the full pre- and post-conditions required for formal methods.

If user-profile information is available, it can define 'usage invariants,' formulas that are frequently preserved in use.

Systems like Daikon and TestEra can be extended to use these ideas in two ways: (1) They can use tests generated from a profile, and adjust their algorithms to generate 'invariants' that may lack generality but do describe usage; (2) They can be given data invariants derived from an external specification, which they then assume for their further operation. Stronger post-conditions can be validated when impossible state values are precluded.

4. REFERENCES

[1] M.D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Soft. Eng.*, pages 99–123, February 2001.

[2] Robert W. Floyd. Assigning meanings to programs. In *Proceedings Symposium Applied Mathematics*, volume 19, pages 19–32. Amer. Math. Soc, 1967.

[3] Matthew M Geller. Test data as an aid in proving program correctness. *Comm. of the ACM*, pages 368–375, May 1978.

[4] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, 1975.

[5] Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 49–58, San Diego, CA, USA, October 8–10, 2003.

[6] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *Proceedings ECOOP '03*, Darmstad, 2003. The authors recommend `www-plan.cs.colorado.edu/henkel` because the proceedings is garbled.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, pages 576–585, October 1969.

[8] D. Marinov and S. Khurshid. Testera: a novel framework for automated testing of java programs. In *Proceedings 16th IEEE Int. Conf. on Automated Software Engineering*, pages 22–34, San Diego, 2001.

[9] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. on Soft. Eng.*, 21:19–31, 1995.

[10] Tao Xie and David Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.

[5] Tao Xie [10] uses the accurate "operational abstraction."