Rethinking the Taxonomy of Fault Detection Techniques

Michal Young
Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine[1] *92717*

Abstract

The conventional classification of software fault detection techniques by their operational characteristics (static vs. dynamic analysis) is inadequate as a basis for identifying useful relationships between techniques. A more useful distinction is between techniques which *sample* the space of possible executions, and techniques which *fold* the space. The new distinction provides better insight into the ways different techniques can interact, and is a necessary basis for considering hybrid fault detection techniques.

Keywords: Fault detection, hybrid analysis techniques, static analysis, dynamic analysis.

1 Introduction

Software validation techniques are usually classified as *dynamic analysis* if they involve program execution, or *static analysis* otherwise. This dichotomy serves a useful purpose in planning validation activities, if fault detection techniques are considered in isolation. But a thorough validation regimen incorporates several fault detection techniques, and it is important to consider their interactions. The static/dynamic distinction is not very helpful in this regard.

Every practical fault detection technique necessarily embodies a delicate balance of accuracy and effort. The design tradeoffs made in achieving this balance are more useful in elucidating relations between techniques and taking advantage of their interactions than the static/dynamic distinction. A particularly useful distinction is between state-space analysis techniques that *fold* actual execution states together (to make the state space smaller or more regular) and those that explore only a *sample* of possible program behaviors. The strategies of folding and sampling result in different sorts of inaccuracy (*pessimistic* and *optimistic*, respectively), which are sometimes erroneously equated with the static/dynamic distinction.

[1] This work was supported in part by the National Science Foundation under grant CCR-8704311, with cooperation from the Defense Advanced Research Projects Agency, by the National Science Foundation under grants CCR-8451421 and CCR-8521398, Hughes Aircraft (PYI program), and TRW (PYI program).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

2 The Conventional Taxonomy

Software modeling and analysis techniques are commonly taxonomized according to their operational characteristics. A central dichotomy in these taxonomies is the distinction between *static analysis,* which does not require program execution, and *dynamic analysis,* which does. For example, the taxonomy of techniques presented in a popular IEEE tutorial [MH81] (see Figure 1) is organized essentially along two dimensions, one being static versus dynamic analysis and the other being the type of documents (requirements, design, or source code) used by the technique.

This conventional taxonomy is well suited to the practical end of planning a series of validation activities in a project. It identifies the sort of documents required for each technique, and thereby allows the project manager to determine where the technique may fit in a project's life cycle. The distinction between static and dynamic analysis also serves this planning purpose, since execution generally requires a piece of completed code. (The whole system is not necessarily required, but if it is not available then some test scaffolding is needed in addition to the modules to be tested).

Sometimes the conventional taxonomy is also taken to indicate the relative cost of techniques, though this is misleading. A common rule of thumb is that static analysis is (computationally) cheaper than dynamic analysis. In fact, either dynamic or static analysis may be expensive or cheap, depending on the thoroughness and accuracy of the particular analysis technique. The characteristics in the proposed extension to the conventional taxonomy, described below, are a more reliable guide to computational cost.

The most significant inadequacy of operational characterization is seen, however, when attempting to formulate an integrated approach to software validation. Recognizing that no single technique is capable of addressing all fault-detection concerns, various attempts to define a comprehensive software validation scheme have been put forth, such as [Ost84] and [Tay84]. The limited success of these attempts is due in part to their static analysis/dynamic analysis orientation. In particular, the operational taxonomy predisposes one to

©1989 ACM 0270-5257/89/0500/0053$00.75

Recommended by: Richard Kemmerer

	Static	Dynamic
Requirements	Informal checklists Formal modeling	Functional testing Testing by classes of input data Testing by classes of output data
Design	Static analysis of design documents	Design-based testing
Programs	General information Static error analysis Symbolic execution	Structural testing Expression testing Data-flow testing

Figure 1: A conventional taxonomy of software modeling and analysis techniques, from [How81b] and [How81a].

view each technique as applied in isolation (or in sequence) and obscures the more substantive concerns of technique interaction. In particular, every modeling or analysis technique involves some compromise between accuracy and completeness on the one hand, and tractability on the other. The dimensions of this tradeoff are largely orthogonal to the issue of whether or not program execution is involved. These tradeoffs are explored in Section 3.

The shortcomings of operational classification are highlighted by the family of techniques known as symbolic execution, symbolic evaluation, or symbolic testing. These techniques do not fit clearly in either the static analysis or dynamic analysis category. Howden [How81b, How77] places symbolic testing among static analysis techniques, although conventional program testing is a special case of symbolic testing. In fact, variations on symbolic execution span the gamut from formal verification to testing. Section 4 describes in more detail the problem of lumping these techniques in the "static analysis" category, and how these problems are avoided by a revised categorization.

3 Analysis Tradeoffs

Since the question "Does program P obey specification S" is undecidable for arbitrary programs and specifications, every fault detection technique embodies some compromise between accuracy and computational cost. It is important to grasp that the necessity of admitting inaccuracy does *not* arise out of limitations in the current state of the art. Rather, since the presence of faults is generally an undecidable property, it is not even theoretically possible to devise a completely accurate technique which is applicable to arbitrary programs. Every practical technique *must* sacrifice accuracy in some way.

The conventional taxonomy of fault detection techniques captures some important dimensions of the analysis technique design space (particularly the relation between sources of information and the classes of faults detected by a technique). However, it does not adequately address tradeoffs between accuracy and computational effort.

Dimensions. The primary means available to reduce the complexity of analysis are *folding* states together (i.e., by abstracting away details) or *sampling* a subset of the state space. Figure 2 summarizes (and considerably oversimplifies) the relationship between effort, folding, sampling, and inaccuracies. These relationships are the subject of the remainder of this section, and are the basis of our revision of the conventional taxonomy of fault detection techniques.

Since the purpose of a taxonomy is practical rather than theoretical, we treat computational effort as a continuum with no sharp division between the truly impossible and the merely hopeless. The "threshold of tractability" in Figure 2, which separates pragmatic techniques from theoretically possible but vastly expensive decision procedures, is more important than the imaginary "threshold of decidability" beyond which effective algorithms do not exist. (A more accurate representation of the design space would depict the "effort" axis stretching off to infinity, with guaranteed proof of correctness and exhaustive testing at infinite distance from the origin. Figure 2 compromises accuracy in order to fit the diagram on a finite page.)

In order to limit computational expense, a technique must admit at least one of two possible kinds of inaccuracy, *pessimistic* inaccuracy or *optimistic* inaccuracy. Pessimistic inaccuracy is failure to accept a correct program. Optimistic inaccuracy is failure to reject an incorrect program. Techniques which admit of pessimistic inaccuracy but no optimistic inaccuracy are sometimes called *conservative*. Optimism and pessimism are shown as opposite directions on a single axis in Figure 2 because most often optimism results from sampling and pessimism results from folding. In principle, and sometimes in practice, a single technique may suffer both sorts of inaccuracy.

Automatic construction of program proofs for arbitrary programs lies at the apex of the design space triangle, representing the ideal case of complete accuracy. Being infallible, it is allowed neither to construct an invalid proof, nor to fail to find a valid proof for a correct program. Such an infallible technique is, of course, impossible (because of the halting problem). The more reasonable hope of automatically constructing program proofs for *some* programs, or assisting a programmer in constructing proofs, is a pessimistic tech-

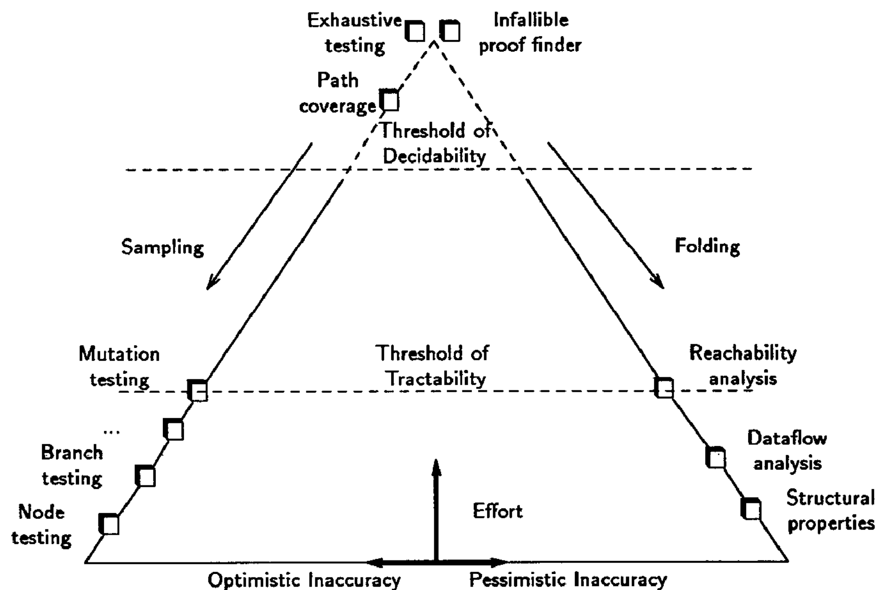

Figure 2: An intuitive view of tradeoffs between accuracy and effort.

nique, because failure to construct a proof does not imply the program is incorrect.

Exhaustive testing shares the apex of the design space with infallible proof construction. Exhaustive testing is, in fact, a "proof by cases" of program correctness. Testing all executable paths through a program (which is already only a sample of the space of behaviors generated by all possible input data) is generally impossible because programs with loops have an infinite number of possible paths.

The reader may find it unintuitive to equate infinite effort for exhaustive testing with undecidability in program proving. To see that these are in fact the same problem seen from different perspectives, consider the following procedure for verification: Generate all theorems derivable, by valid proof rules, from axioms describing the program. This procedure is easily mechanizable. Syntactic variations can be generated in the same manner, by including appropriate rewrite rules among the valid proof rules. If any of the generated theorems is identical to the program specification, announce that the program has been verified and halt. Clearly, this procedure would be sound *if* one could wait forever (literally) for an answer. Exhaustive testing is just the same.

Models of Execution. We have ignored (and will continue to ignore, for the most part) another difference between proving and testing. Notions of correctness, reliability, etc., are only meaningful with reference to a model of computation. Two rather different models are used in testing and proving, and they result in quite different notions of what validation means.

A theoretical, ideal machine perfectly obeys the semantics of a particular language in executing a program. This idealized machine may be quite different from any physical machine. The theoretical machine never has roundoff error or arithmetic overflow, or runs out of storage, or incorrectly interprets a program. Correctness with respect to such a model does not imply that a program will execute correctly on all, or even some, physical machines, unless the machine (including its compiler, operating system, etc.) has also been verified.

The alternative is to refer to a particular target machine on which a program is to run. This has the advantage that all levels of implementation may be validated at once. On the other hand, particular machines can have peculiarities which allow a program to run "correctly" on that machine, although it will fail on most other machines (and certainly on the ideal machine, which makes no promises except to obey the semantics of a language).

It is convenient to assume that the ideal machine is an accurate model of a particular physical machine (or vice versa) when comparing or combining fault detection techniques. Moreover, tradeoffs involving effort and accuracy are largely orthogonal to the particular model of computation used. Accordingly it will not be considered further in this paper.

3.1 Folding

An analysis technique may abstract away some details of program execution, either because they are of no consequence or because removing them makes the model easier to analyze. Often, ignoring or removing details has the effect of *folding* states. That is, the technique uses a smaller state space than the actual program, and each state in the model state space represents several states in the normal program execution. In fact, if exhaustive enumeration of the state space is to be practical, a finite number of model states must represent an infinite number of program states.

Usually one wishes to guarantee that simplifications employed in analysis introduce only pessimistic inaccuracy, i.e., that no errors will be hidden. Such a guarantee can only be made relative to a class of specifications. For safety properties of individual states, it is sufficient to show that each potential "bad" state in normal[2]program execution is represented by a "bad" state in the folded model. To preserve violations of specifications regarding paths in the execution state space, including liveness properties and precedence properties, additional conditions must be imposed on the mapping. A set of sufficient conditions for showing that a folding preserves violations of specifications expressed in propositional temporal logic are given in [You88].

In techniques based on program texts, or information derived from program texts such as flowgraphs, the degree of folding will generally be determined by the class of model. For instance, many techniques model control flow and omit data, thus folding together program states which differ only in variable values.

It is also possible to fold instances of a model within the same class. For instance, in analyzing concurrent systems, a principle sometimes called "virtual coarsening" allows many sequential steps of a process to be combined when those steps are independent of other processes. Applications of virtual coarsening include reductions of control graphs by analysis tools in the SARA design environment [EFRV86], and reductions of program flowgraphs in the anomaly detection techniques of Taylor [Tay83] and Long [LC89].

Folding in program verification. In formal program verification one usually avoids explicitly constructing representations of program states. Instead, theorems describing properties of program behavior are derived. Since the set of theorems derivable from a program text taken together with a set of axioms and rules of inference expressing the semantics of a programming notation[3] is infinite, exhaustive enumeration of theorems is no more practical than exhaustive enumeration of execution states. The hierarchical structure of abstractions necessary for practical verification is exactly mirrored by a set of foldings of the execution state space, even when no explicit representation of the execution state space is constructed.

Consider a proof of a program involving an abstract data type (ADT). It is impractical to consider the implementation details of the ADT while verifying properties of the program as a whole. Instead, one first verifies properties of the ADT, and then verifies properties of the program as a whole in terms of the abstraction. A typical program verification will involve many steps of this type; a verification performed completely at the level of implementation details would be incomprehensible.

A representation invariant for an abstract data type folds the state space of the implementation into the state space of the data abstraction. Proof of the representation invariant is required to justify reasoning in terms of the folded state space. The creative part of verification is typically choosing the right abstractions, i.e., determining how to fold the space to make the next inference step easy.

When an abstraction step is generally useful and does not require creativity to apply, it may be embodied in a completely mechanical procedure. Whereas formal verification involves creative processes, including many repeated foldings of the execution state-space, automatic techniques such as data flow analysis or reachability analysis apply a single folding. The soundness of this folding must be established once and for all in justifying the analysis technique. A simple example is static type checking. Using formal verification techniques, it would be possible but challenging to prove that a particular Lisp program, with no type declarations, never attempted to take the tail of an integer nor divided a list by a floating point number. By requiring that each variable take on values of only a single, declared type, strongly typed languages make the verification simpler — so simple that it can be completely mechanized and routinely carried out by compilers.

3.2 Sampling

A simple strategy for dealing with an infinite state space is to explore only part of it. Many techniques do just this. In particular, since the set of actual computation states of most programs is infinite,[4] all techniques normally grouped under the rubric of *dynamic analysis* sample the state space of program execution.

The well-worn admonishment that program testing can reveal the presence of errors but not their absence derives from the fact that a sampling technique may admit optimistic inaccuracy. Pessimistic inaccuracy in a sampling technique is impossible unless normal execution is disturbed in some way (e.g., if execution of a real-time program were slowed enough to miss a deadline). Much of the testing literature, therefore, is concerned with determining when a sufficiently representative portion of the state space has been explored to merit some confidence in the unexplored portions. (Contrary to what Figure 2 may suggest, common coverage metrics are only partially ordered with respect to optimistic inaccuracy; see, for instance, [CPRZ85] and [FW86].)

[2]*Normal* execution may be execution on an ideal, theoretical machine, or execution on a particular machine, depending on the goals of analysis. When there is a mismatch between the goal and the "normal" model, both pessimistic and optimistic inaccuracies can result. For instance, if one is interested only in execution on a particular machine, a deadlock detection technique based on the semantics of a programming language may both report errors which cannot occur because of peculiarities of the process scheduler on the target machine (pessimism) and fail to uncover errors in the implementation of the language (optimism).

[3]This discussion is oriented toward axiomatic semantics and verification in the Floyd-Hoare style. Details of the relation between implicit and explicit representations of an execution state-space would differ if one chose, say, a denotational framework, but a similar relation could be drawn.

[4]It is actually possible to fix a finite bound on the state space of program executions on any particular hardware. Since the number b of bits of storage available to a program, must be finite (infinite tapes exist only in theoretical machines), the program can be in only 2^b distinct states. Even for small machines this number is greater than the number of nanoseconds since the big bang, so it is practical to consider it infinite.

Sampling is not limited to techniques that explore the state space of normal program execution (conventional testing). All varieties of symbolic execution, for instance, fold states together by representing a large number of actual data states by a smaller number of symbolic data states. Many varieties of symbolic execution explore only a portion of the resulting state space, because although "smaller" it is generally still infinite. (Varieties of symbolic execution are considered in more detail in Section 4.) Some models with finite but large state spaces also rely on sampling. For instance, a Petri net may be exhaustively analyzed in some cases, but when exhaustive analysis is impossible, Petri net simulation may be used [Raz87].

Choosing samples. In classical testing of programs, program behavior is controlled by input data. Thus, a sample of program behaviors is chosen indirectly through a choice of test data. Uniform sampling of the input space (random testing) generally projects onto a very non-uniform sample of the space of possible execution states; consider:

```
if i = j then
   Do something wrong
else
   Do the right thing
end if;
```

If i and j are integers, and are inputs to the program, then random selection of inputs has an infinitessimal chance of exercising the erroneous behavior.

Test adequacy criteria are designed to insure that the behaviors chosen are appropriately distributed to increase the likelihood of revealing errors. Often this is accomplished by partitioning behaviors into classes, and requiring samples to be drawn from each class. Specification-based selection criteria apply directly to test data, but explore different parts of the state space (or reveal missing program logic) by testing classes of data that must be treated differently. Control flow and data flow coverage criteria [RW82, CPRZ85, FW86] relate more directly to the state space. These structural criteria for sampling can be related to a folded model of execution, i.e., the coverage criterion is satisfied if each state in the folded model is represented by at least one state in the sample.

Hamlet pointed out in [Ham87] that an appropriate distribution depends on the purpose of testing. For reliability estimates, an operational distribution of inputs is required (but often impossible to obtain). For confidence in "probable correctness," appropriate samples must be drawn from the space in which faults are distributed, which is more closely related to the textual space of the program. The results of probable correctness theory are so far mostly negative; [HT88] shows that partitioning (which includes all functional and structural coverage criteria) is not an improvement over random testing for achieving high statistical confidence in program correctness. On the other hand, partitioning may be as close to uniform sampling as one can achieve in practice.

An alternative to both reliability estimates and probable correctness is confidence in the absence of particular faults. Several fault-based and error-based criteria for selecting test data have been put forward. Mutation testing is most direct: A sample of program behaviors is judged adequate when it differentiates between the actual program and a set of programs with hypothetical faults. Other fault-based methods avoid explicitly creating alternative programs, but attempt to select test data that would reveal a particular class of faults if it were present.

4 Example: Symbolic Evaluation

The clearest example of the inadequacy of the conventional static/dynamic dichotomy is the group of techniques known collectively as symbolic evaluation. These techniques are conventionally classified as static analyses, because they do not involve normal program execution. Two symbolic evaluation techniques which share a representation of a program state, and very little else, are described below. The differences between these techniques, their capabilities, and their shortcomings illustrate the problems inherent in lumping them together in a taxonomy of fault detection techniques. The revised taxonomy reveals that, while both techniques employ some folding, one folds the state space further to allow exhaustive enumeration of program behaviors, and the other visits only a sample of the complete space of possible states.

The two techniques described here are *symbolic execution* and *global symbolic evaluation.* The description of symbolic evaluation methods here differs in detail from descriptions in the literature, in order to simplify the presentation and highlight the state-space perspective. We limit attention to programs containing only assignment statements, *while* loops, and *if* statements, and ignore procedure calls and input/output.

4.1 Symbolic execution

The model schema used by symbolic execution is a program flowgraph, with nodes for each executable program statement. An additional node is placed before the first executable statement, and one after each terminal node. *If* statements and *while* loops are represented by nodes with two out-edges. A token is used to represent a thread of control. (For the current discussion, we assume a single thread of control.) Two additional pieces of information are maintained. The *path expression* associates program variables with symbolic values (algebraic expressions). The *path condition* is a predicate which describes the conditions necessary to follow a particular execution path.

Symbolic execution begins with a token on the edge leading into the first executable statement of the program. (We added a node before this statement so that execution could begin with a token on this edge.) The path condition is initially set to *true,* and the path expression associates each program variable with a unique symbol.

Analysis proceeds by advancing the token through a statement, and onto an edge leaving the statement. When a token is advanced through an assignment statement, the path expression is modified. For instance, if the assignment were $C := A + B$, then the current expression associated with C would be replaced by $\alpha + \beta$, where α and β are the current symbolic values of A and B, respectively.

Advancement of a token through a conditional branch node (*if* or *while*) adds a term to the path condition, corresponding to the branch chosen. For instance, if the branch condition were $A = B$, and the *true* branch were chosen, then $\alpha = \beta$ would be conjoined with the path condition, where α is the expression associated with A in the path expression, and β is the expression associated with B. If the *false* branch were chosen, then the complement of that predicate would be conjoined with the path condition. If the new path condition is inconsistent (provably equivalent to *false*), then the path is unexecutable.

States in symbolic execution are characterized completely by *(path expression, path condition)* pairs. (If nondeterministic choice were possible, the current token location would also be required.) For a program without loops, this state space will be a tree that could, in principle, be exhaustively explored by a reachability analysis technique.

For programs with loops, the state space will generally be infinite. Thus, exhaustive generation of states is ruled out. Instead, sampling can be used to explore only a portion of the state space. Symbolic execution starting from the initial state and progressing along some path to a terminal state is called *symbolic testing.*

Symbolic testing is like conventional program testing in most ways. It admits the same sort of optimistic inaccuracy (since faults may lie on paths which are not explored), requires an oracle ("Is the final path expression acceptable?"), and like conventional program testing may be practiced to some coverage criterion.

4.2 Global symbolic evaluation

Whereas symbolic execution explores a sample of the state space, global symbolic evaluation folds the infinite state space of symbolic evaluation into a finite set of representative states. Thus, while symbolic execution is like conventional dynamic analysis techniques, global symbolic evaluation is like other static analysis techniques.

Global symbolic evaluation attempts to fold together states reached along paths which differ only in the number of iterations a loop is traversed. One approach to global evaluation is to represent the effect of a loop by a set of recurrences, which in some cases can be simplified to closed form expressions. An alternative, especially suitable when symbolic evaluation is used as an aid to formal verification, is to cut each loop with an assertion. The latter approach is discussed here.

Imagine that every loop in a program is cut by a loop invariant assertion which lies on a flowgraph edge. Also, an edge leading into the first program statement is labeled with an assertion describing the domain of applicability of the program, and the edge leading out of each terminal statement is labeled with an assertion stating the required output condition of the program. An assertion is satisfied if, for every state in the state space of symbolic evaluation such that the token lies on an edge labeled by the assertion, the assertion can be proven from the path condition and path expression. If the path expression and path condition are not sufficient to prove the assertion, then we say the assertion is violated.

If every loop is cut by assertions, then every path through the flowgraph is made up of a sequence of subpaths between assertions. Every such subpath is finite, and there are a finite number of them. Analysis proceeds as for symbolic testing, but with one important difference: instead of following a path from the beginning of execution, each subpath from one assertion to the next is separately analyzed. For each such subpath, the path expression and path condition are initialized to reflect the initial assertion. At the final state along each subpath, the final assertion is checked against the path condition and path expression. If the assertion cannot be proven, an error is reported. If each individual subpath is accepted, then every path through the program must satisfy every predicate.

Note that what the loop-cutting assertions have done is to fold infinite sets of states into representative states by discarding details of the execution state. Each time an assertion is reached at the end of a subpath, the path expression and path condition may be different. Details of these different conditions are discarded, and only the information in the assertion is preserved. One may think of the assertions as filters which prevent too much information from passing through.

Unlike symbolic testing, global symbolic execution is a *pessimistic* technique. It does not accept incorrect programs (assuming, of course, that the input and output assertions fully and correctly capture program specifications). Pessimistic inaccuracy stems from the difficulty of finding satisfactory loop-cutting assertions. An unsatisfactory assertion will either be too strong (not provable at terminal states along partial paths) or too weak (not sufficient as an assumption to prove the next assertion along a subpath), or perhaps both. In fact, since successful global symbolic evaluation using the loop-cutting method is a machine-aided proof of partial correctness using the loop invariant method [HK76, KE85], finding satisfactory assertions is an unsolvable problem in the general case.

4.3 Summary

Both symbolic testing (symbolic execution of particular program paths) and global symbolic evaluation employ some folding, namely representing whole classes of data by symbolic values. This degree of folding still leaves an infinite state space which cannot be exhaustively explored. Symbolic testing examines only a finite sample of this space, and thus incurs optimistic inaccuracy. Global symbolic evaluation folds the state space further, to obtain a finite number of representative states, and thus incurs pessimistic inaccuracy.

A taxonomy intended to facilitate combining analysis techniques in an integrated validation regimen must highlight the relative strengths and weaknesses of each technique. In the case of symbolic evaluation techniques, it should point up the optimistic inaccuracy of symbolic testing (and its similarity to conventional testing in that regard) and the pessimistic inaccuracy of global symbolic evaluation. A revised taxonomy that does just that is introduced in the following section.

5 A Revised Taxonomy

5.1 The taxonomy

The following taxonomy highlights the fundamental characteristics of state-space analysis techniques from the viewpoint of considering how they may be combined. The primary characteristic, not surprisingly, is the distinction between

- *state folding* and
- *state sampling.*

Additionally, the characteristics of

- *model schemata,*
- *representation of the state space*, and
- *oracle*

are called out, because of their practical utility in considering technique combination.

Having discussed folding and sampling at length, we briefly consider the three auxiliary characteristics.

Model schemata. A class of model schemata (Petri nets, flowgraphs, program texts in some language) determines a class of state spaces within which sampling or folding may occur. It is easiest to exploit interactions between techniques when the same model schemata is shared between them. For instance, Young and Taylor [YT88] describe a method for combining static concurrency analysis with symbolic execution, based on the observation that they share an underlying flowgraph model of execution, and that the state space of the former is (conceptually) obtained from the latter by folding together states with different path expressions (data values).

Representation of the state space. Some techniques explicitly represent the state space in the form of a *reachability graph,* while other techniques leave the state space implicit and represent only a single "current" state at any one time. Some techniques (notably test coverage metrics) keep a partial record of the portions of the state space visited. Some techniques, e.g. constrained expression analysis [ADWR86], infer properties of a state space without constructing any explicit representation of it.

Oracles. Among techniques which build explicit representations of a state space (whether partial or complete), one can often distinguish a component of the technique for exploring the state space from a procedure for determining whether a state or path is faulty. In the testing literature, for instance, coverage criteria are usually treated separately from test oracles. The notion of oracle is present in practically every technique, however, and so is called out here. Important characteristics of oracles include:

- Is an oracle function explicit in the technique? Some techniques (notably test coverage metrics) assume the availability of an oracle, but do not describe how to build it.
- If the oracle is explicit, does it check individual states or paths? If it checks states, does it check only a subset of states (e.g., terminal states)?
- Does the oracle check for certain fixed properties (e.g., absence of deadlock) or may a class of properties be specified by the user? (When fixed properties are checked, these are usually *implicit specifications,* whereas explicit specification are supplied by the user.)

5.2 Sample derivative categories

These characteristics can be used to divide state space analysis techniques for fault detection into groups exhibiting similar properties in the critical dimensions. Some, which seem to us to arise naturally, are flow analysis, reachability analysis, and testing. They are considered in turn below, in the light of the taxonomy's characteristics. The purpose of this section is not to give a rigorous grouping; it is just to illustrate how some familiar techniques appear in the light of the revised taxonomy.

Flow analysis.

Folding: Control flow is modeled, data values are ignored. Sampling: None.

Model schema: The model schema is a directed graph that can itself be considered as a state space, and the analysis procedure labels nodes in this graph with predicates. We include in this class both techniques derived from classical data flow analysis, e.g. [OO86], and other techniques which similarly label a directed graph, such as the temporal logic checking algorithms of Clarke, Emerson, and Sistla [CES86] and Fernandez, Richier, and Voiron [FRV85]. These techniques are procedures for *checking* a graph rather than for building it, so they may naturally be combined with other techniques for constructing a representation of the state space.

Representation of the state space: Explicit, given as input (the control graph).

Oracle: Hard-wired classes of anomalous behavior in the case of systems like DAVE [OF76], programmable and path-oriented in [OO86].

Reachability analysis.

Folding: Control flow is modeled, data values are ignored.

Sampling: None.

Model schema: Various graph models, including Petri nets and program flowgraphs.

Representation of the state space: Finite, explicitly represented, and exhaustively enumerated (generated). Examples of reachability analysis include Petri net reachability analysis [Pet81], static concurrency analysis [Tay83], and analysis in the "control" domain of UCLA graph models [EFRV86]. Because reachability analysis is exhaustive, all reachability analysis techniques *fold* the space of program behaviors into a finite state space, and thus tend to be *pessimistic*. These are primarily techniques for constructing a representation of a state space.

Oracle: Although a procedure for checking particular properties may be hard-wired into a reachability analysis technique, it is usually superior to employ a logically separate, "programmable" checking procedure. An example of this approach is the P-Nut system of Razouk and Morgan [Raz87, MR87], which combines Petri net reachability analysis with a checking procedure for branching time temporal logic assertions.

Classical Testing.

Folding: Classical program testing explores the state space of actual program execution with no folding.

Sampling: The state space is infinite or very large, and only a portion of it is explored. Usually a *coverage criterion* is specified to provide a way of determining that an adequate sample of behaviors has been inspected. Coverage criteria include path related criteria (used, e.g., in structural coverage schemes) and input/output class criteria (used, e.g., in functional testing).

A coverage criterion may be based on an auxillary, folded model of execution. For instance, data flow testing [RW82, CPRZ85] measure representativeness in terms of coverage of certain control-flow subpaths, thereby relating program execution to paths in a flowgraph model.

Model schema: Some form of program text (such as compiled binary).

Representation of the state space: Only the current state is fully represented during exploration.

Oracle: Testing techniques focusing on exploration of the state space, such as structural and functional coverage schemes, usually leave the oracle unspecified. Assertion-checking schemes (e.g. Anna [LvH85]) are programmable state-oriented oracles; TSL [LHM+87] is programmable and path-oriented.

Folded testing.

Folding: Classes of data are folded in the case of symbolic testing. Additional folding of implementation details may occur in simulations based executable specifications such as Petri nets or PAISley [ZS86]. In these techniques, the state space is considerably simplified by comparison to actual program execution, but may still be too large to exhaustively enumerate.

Sampling: Exploring a sample of the behaviors of an abstract model of execution is clearly a testing technique (and subject to the same optimistic inaccuracy), even when conventional program code is absent as in executable specifications. While the notion of coverage criteria has usually been applied only to testing of actual execution, it is applicable as well to other models of execution. But whereas classical testing selects a sample of the execution space indirectly by selecting test data, non-determinism in abstracted models may allow more direct sampling of the execution state space. Executable specification systems typically support random choice in simulation. The Argos system for protocol testing also allows heuristic guidance [Hol87].

6 Combining Fault Detection Techniques

Taxonomies are created for practical ends. The conventional taxonomy, as presented in [MH81] and elsewhere, is organized in a manner that makes it very useful for test planning, since a major axis of the taxonomy is the type of documents used by each technique. But the conventional taxonomy is not very useful as a guide to devising new techniques, because the distinction between static and dynamic techniques does not capture enough of the tradeoffs involved in designing such a technique. In this section we argue that the extended taxonomy can provide guidance for devising hybrid analysis techniques and integrated approaches to software fault detection (by highlighting the nature and extent of inaccuracy of each technique potentially employed). We illustrate this by discussing some general directions for integration based on the characteristics in Section 5, and showing how some existing approaches fit in this framework.

6.1 Using pessimistic techniques to concentrate optimistic techniques

Pessimistic techniques often fail to distinguish between executable and non-executable paths. If the state space of a pessimistic technique can be related to the state space of an optimistic technique, it may be possible to concentrate the optimistic technique on just those portions of the state space which are reported faulty by the pessimistic technique. For instance, if a class of faults can be ruled out by using a pessimistic technique like flow analysis, then optimistic techniques like testing should concentrate on finding other faults. If symbolic evaluation with loop-cutting assertions is used to show that most of the assertions in a program are satisfied, testing should be concentrated on paths that pass through the assertions that cannot be verified. An integrated validation methodology which uses pessimistic techniques to concentrate optimistic techniques has been described in detail by Osterweil [Ost84].

6.2 Combining pessimistic techniques

Two pessimistic techniques may also benefit from combination, if one is more pessimistic (folds the state space farther) than the other. As mentioned earlier, [YT88] proposes combining static concurrency analysis with symbolic execution on this principle. The less pessimistic technique need only follow paths that lead the more pessimistic technique to errors; an error is reported only if both techniques reach it. One can imagine analogous approaches for techniques based on other model schemas. For instance, it is not difficult to show that interpreting a Time Petri net [Mer74] as a standard (untimed) Petri net is a more pessimistic approach than interpreting the time information.[5] That is, the untimed interpretation will reach every state reachable by the time version. If analyzing the standard interpretation is cheaper (because it folds together states that differ only with respect to remaining enable time), it could be used to guide the timed interpretation.

6.3 Combining optimistic techniques

A technique that attempts to raise confidence about the whole space of program behaviors by exploring a portion of the state space depends implicitly on a claim that the portion explored is *representative* of other portions. A coverage criterion identifies classes of states or paths which are similar in some way, so that any member of a class can be taken as a representative of other members of the same class. Confidence hinges on the proposition that if one path in a class is faulty, other paths in the same class are likely to be faulty also, and so exploring one path in the class is likely to uncover a fault. In principle, therefore, one should attempt to identify classes of paths which are treated identically in all important respects. A step toward this ideal is to combine coverage metrics which group paths by different criteria, e.g., functional and structural testing. A technique which combines different coverage criteria in this way is partition analysis.

Partition analysis. Richardson and Clarke have described a hybrid fault detection technique which illustrates some of the principles described above [RC85]. The input domain of a program (and thus, paths through the program state space) are partitioned according to treatment by the specification and, independently, according to treatment by the implementation. The implementation partition is derived by symbolic execution, so that each class of paths in the implementation is represented by a *(path condition, path expression)* pair. The specification partition (classes of inputs treated the same in the specification) is represented similarly as a set of *(condition, computation)* pairs.

[5] The standard interpretation will contain a sequence of firings for every sequence of firings in the time interpretation. This will make it pessimistic for most classes of errors, but it is possible to compose a branching time temporal logic assertion which may be satisfied only by the standard interpretation, e.g., an assertion that a certain state is reachable but not inevitable.

The specification partition and implementation partition are combined by considering every combination of classes, i.e., the Cartesian product of the two partitions. For each pair in the combined partition, the respective path conditions are conjoined. If the combined path condition is inconsistent, the class is empty. Otherwise, proof techniques are used to show that the path expression derived from the implementation is equivalent to the computation required by the specification.

Verification of each combined non-empty class of paths is followed by testing. Since proving that path expressions are equivalent to required computations is pessimistic, testing can concentrate on those partitions where proving fails. But in contrast to the strategy described above for combining pessimistic and optimistic techniques, partition analysis calls for testing even when verification is successful. Testing is called for because of potential differences between the implementation of a program on a real machine and the interpretation of path expressions in terms of an ideal, theoretical implementation.

7 Conclusion

The dichotomy between static and dynamic analysis in the conventional taxonomy of fault detection techniques [MH81] is too coarse a distinction to serve as a guide for combining techniques and devising new, hybrid fault detection techniques. We have proposed to partially remedy that situation by replacing the static/dynamic distinction by a distinction between *sampling* the space of possible behaviors, and *folding* states together to make the space smaller. This distinction is better at capturing important tradeoffs in the design of state-space analysis techniques.

All practical techniques are vulnerable to some kind of inaccuracy. The distinction between *pessimistic* inaccuracy (characteristic of techniques which limit effort by folding states together) and *optimistic* inaccuracy (characteristic of techniques which explore only a sample of a state space) is the major dimension of the extended taxonomy. In many cases this distinction coincides with the static/dynamic dichotomy: Most static analysis techniques are pessimistic, and all dynamic analysis techniques are optimistic. The folding/sampling distinction, though, is not just a new name for the old dichotomy. This is shown most clearly by the case of symbolic evaluation, which is conventionally considered a static technique, but which actually encompasses both folding and sampling techniques.

By focusing on analysis tradeoffs, and especially on the relation between strategies for state-space exploration and inaccuracy in the pessimistic or optimistic direction, the revised taxonomy suggests some fruitful areas of research. For instance, coverage criteria are applicable in principle to any sampling technique, and not just to conventional program testing. And while coverage criteria, which relate the extent and nature of sampling to the extent of optimistic inaccuracy, is an active research topic, there is currently no analogous theory characterizing the relation of folding to pessimistic inaccuracy.

Finally, a taxonomy which recognizes the design tradeoffs inherent in devising fault detection techniques is a useful guide to the potential interactions between techniques. This should become increasingly important in the future, as more research moves beyond consideration of individual techniques applied in isolation, to integrated application of combinations of techniques and new hybrid techniques.

References

[ADWR86] George S. Avrunin, Laura K. Dillon, Jack C. Wileden, and William E. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Trans. Software Eng.*, SE-12(2):278–292, February 1986.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Trans. Programm. Lang. Sys.*, 8(2):244–263, April 1986.

[CPRZ85] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proc. 8th Int. Conf. Software Eng.* , pages 244–251, London, August 1985. ACM Sigsoft.

[EFRV86] Gerald Estrin, Robert S. Fenchel, Rami R. Razouk, and Mary K. Vernon. SARA (System ARchitects Apprentice): Modeling, analysis, and simulation support for design of concurrent systems. *IEEE Trans. Software Eng.*, SE-12(2):293–311, February 1986.

[FRV85] J. C. Fernandez, J. L. Richier, and J. Voiron. Verification of protocol specifications using the CESAR system. In *Proc. 5th Int. Workshop Protocol Specification, Testing, and Verification*, Toulouse, France, June 1985.

[FW86] Phyllis G. Frankl and Elaine J. Weyuker. Data flow testing in the presence of unexecutable paths. In *Proc. 1st Workshop on Software Testing*, pages 4–13, Banff, Canada, July 1986. ACM/Sigsoft and IEEE-CS.

[Ham87] Richard G. Hamlet. Probable correctness theory. *Information Processing Letters*, 25:17–25, April 1987.

[HK76] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

[Hol87] Gerard J. Holzmann. Automated protocol validation in *argos*: Assertion proving and scatter searching. *IEEE Trans. Software Eng.*, SE-13(6):683–696, June 1987.

[How77] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. Software Eng.*, SE-3(4):266–278, July 1977.

[How81a] William E. Howden. A survey of dynamic analysis methods. In E. Miller and W.E. Howden, editors, *Tutorial: Software Testing & Validation Techniques*, pages 209–231. IEEE Computer Society Press, 1981. Second Edition.

[How81b] William E. Howden. A survey of static analysis methods. In E. Miller and W.E. Howden, editors, *Tutorial: Software Testing & Validation Techniques*, pages 101–115. IEEE Computer Society Press, 1981. Second Edition.

[HT88] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. In *Proc. 2nd Workshop on Software Testing, Analysis, and Verification*, pages 206–215, Banff, Canada, July 1988. ACM/Sigsoft and IEEE-CS.

[KE85] Richard A. Kemmerer and Steven T. Eckmann. UNISEX: a Unix-based symbolic executor for Pascal. *Software — Practice & Experience*, 15(5):439–458, May 1985.

[LC89] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proc. 11th Int. Conf. Software Eng.* , Pittsburgh, May 1989.

[LHM+87] David Luckham, David Helmbold, S. Meldal, Doug Bryan, and M.A. Haberler. Task sequencing language for specifying distributed Ada systems — TSL-1. Technical Report CSL-TR-87-334, Computer Systems Laboratory, Stanford University, July 1987.

[LvH85] David C. Luckham and Friedrich W. von Henke. An overview of ANNA, a specification language for Ada. *IEEE Software*, 2(2):9–22, March 1985.

[Mer74] Philip M. Merlin. *A Study of the Recoverability of Computing Systems*. PhD thesis, Dept. Information and Computer Science, U. California, 1974.

[MH81] Edward Miller and William E. Howden. *Tutorial: Software Testing & Validation Techniques*. IEEE Computer Society Press, second edition, 1981.

[MR87] E. Timothy Morgan and Rami R. Razouk. Interactive state-space analysis of concurrent systems. *IEEE Trans. Software Eng.*, SE-13(10):1080–1091, October 1987.

[OF76] Leon J. Osterweil and Lloyd D. Fosdick. DAVE – a validation, error detection, and documentation system for FORTRAN programs. *Software — Practice & Experience*, 6:473–486, 1976.

[OO86] Kurt M. Olender and Leon J. Osterweil. Specification and static evaluation of sequencing constraints in software. In *Proc. 1st Workshop on Software Testing*, pages 14–22, Banff, Canada, July 1986. ACM/Sigsoft and IEEE-CS.

[Ost84] Leon J. Osterweil. Integrating the testing, analysis, and debugging of programs. In Hans-Ludwig Hausen, editor, *Software Validation*, pages 73–102. North-Holland, 1984.

[Pet81] J. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[Raz87] Rami R. Razouk. A guided tour of P-NUT. Technical Report 86-25, U. California, 1987.

[RC85] Debra J. Richardson and Lori A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Trans. Software Eng.*, SE-11(12):1477–1490, December 1985.

[RW82] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *Proc. 6th Int. Conf. Software Eng.* , pages 272–278, Tokyo, Japan, September 1982.

[Tay83] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM*, 26(5):362–376, May 1983.

[Tay84] Richard N. Taylor. Analysis of concurrent software by cooperative application of static and dynamic techniques. In Hans-Ludwig Hausen, editor, *Software Validation*, pages 127–137. North-Holland, 1984.

[You88] Michal Young. How to leave out details: Error-preserving abstractions of state-space models. In *Proc. 2nd Workshop on Software Testing, Analysis, and Verification*, pages 63–70, Banff, Canada, July 1988.

[YT88] Michal Young and Richard N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Trans. Software Eng.*, 14(10):1499–1511, October 1988.

[ZS86] Pamela Zave and William Schell. Salient features of an executable specification language and its environment. *IEEE Trans. Software Eng.*, SE-12(2):312–325, February 1986.