Rethinking the Taxonomy of Fault Detection Techniques

Michal Young

Software Engineering Research Center
Department of Computer Sciences
Purdue University

Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine

September 1991

Abstract

The conventional classification of software fault detection techniques as static or dynamic analysis is inadequate as a basis for identifying useful relationships between techniques. A more useful distinction is between techniques that *sample* the space of possible executions, and techniques that *fold* the space. The new distinction provides better insight into the ways different techniques can interact, and is a basis for considering hybrid fault detection techniques including combinations of testing and formal verification.

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

An earlier version of this paper appeared in *Proceedings of the 11th International Conference on Software Engineering,* Pittsburgh, May 1989. Address correspondence to the first author at Department of Computer Sciences, Purdue University, West Lafayette, IN 47907. Email: young@cs.purdue.edu. Phone: (317) 494-6023.

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, CCR-8521398, and CCR-9010135, and by Hughes Aircraft (PYI program) and TRW (PYI program).

Contents

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 developing techniques are more useful in elucidating relations between them 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.

Scope. The discussion is limited to techniques aimed at discovering faults[1], primarily in the context of developing and improving software products rather than measuring their quality. We focus on *state-space analysis* techniques, i.e., techniques that explicitly construct some representation of program states. This class of techniques includes conventional testing, in which actual program execution is used to construct program states, as well as a variety of techniques that construct and inspect more abstract models of an execution state space, such as data flow analysis and reachability analysis. We also consider how state-space analysis techniques are related to, and may be combined with, verification techniques that emphasize formal reasoning and suppress explicit description of states and events.

The paper is organized as follows: Section 2 reviews the conventional division of fault detection techniques into *static* and *dynamic* analysis. Section 3 considers inherent tradeoffs between accuracy and computational ef-

[1]Software may contain *faults,* which may cause *errors* in executions of that software, possibly manifested in *failures* in the system of which the software is part. The terms *fault* and *error* are sometimes used in these senses, and sometimes interchanged, in the testing literature. For most of this paper, the distinction between faults and errors is not essential.

fort in fault detection, and proposes classifying state-space analysis techniques according to how they make those tradeoffs. Section 4 uses the example of symbolic evaluation techniques to illustrate the advantage of the sampling/folding distinction over the conventional static/dynamic dichotomy. Section 5 describes a revised classification scheme in which the folding/sampling distinction plays a central role, and applies the revised scheme to some well-known techniques to show that it makes relations among techniques and possible interactions clearer. Sections 6 describes strategies for combining fault detection techniques based on the revised classification scheme. Section 7 considers formal verification techniques and their relation to state space analysis techniques, and discusses two techniques that combine aspects of both. Section 8 concludes.

2 The Conventional Taxonomy

Software modeling and analysis techniques are commonly classified 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 artifacts (requirements documents, 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 information 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 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.

	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 [How81a] and [How81b].

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. The operational taxonomy predisposes one to 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 the static/dynamic dichotomy 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. Adrion, Branstead, and Cherniavsky note the ambiguity of the classification in [ABC82]. 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 that 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. Most fault detection techniques can be viewed as constructing a representation of possible program executions and comparing this representation to a specification of intended behavior. In principle either the representation of possible behaviors or the comparison of actual to intended behavior must be imperfect. A large class of techniques can be characterized as state-space analysis, because actual behavior is represented as (some abstraction of) program states and state transitions. This class includes conventional testing (construction of actual program states in full detail) as well as a variety of techniques that construct more abstract state-transition models.[2]

[2]The distinction between state-space analysis and other analyses is not absolute. Just as a single programming language can be given equivalent operational, axiomatic, and denotational semantics, even analysis techniques without an operational state-space flavor can be interpreted as state-space analyses. The state-space framework is very natural for testing and for a variety of other techniques with a strong operational flavor. Techniques employing data flow analysis and Floyd/Hoare-style verification can be made to fit with only a little stretching. Verification in the Dijkstra/Gries calculational style, which puts a premium on suppressing any residue of operational thinking, would be very awkward to

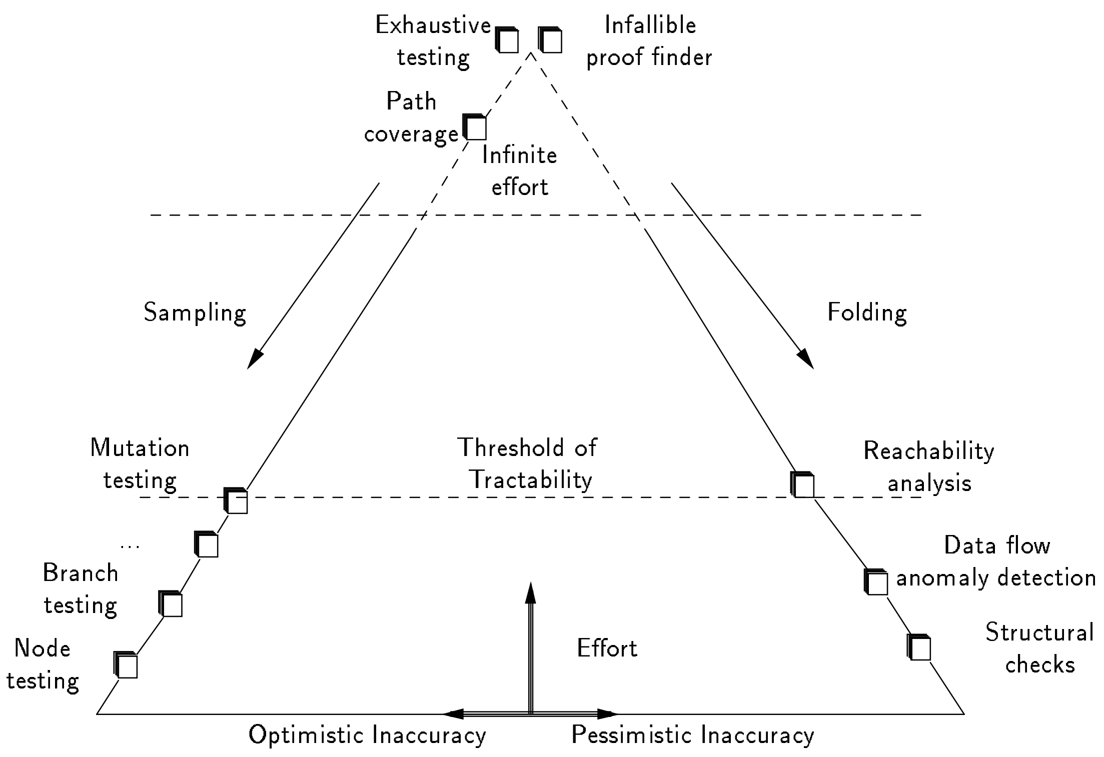

Figure 2: An intuitive view of tradeoffs between accuracy and effort.

The primary means available to reduce the complexity of state-space analysis are *folding* states together or *sampling* a subset of the state space. Informally speaking, folding replaces the complete state-space by a smaller or more regular model, usually by abstracting away some details of execution states. Sampling investigates some portion of the state space, for instance by executing a program on selected input data. 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 tractabil-

characterize as state-space analysis.

ity" in Figure 2, which separates pragmatic techniques from theoretically possible but vastly expensive decision procedures, is more important than the line 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 (strictly speaking, failure to accept a *(specification, program)* pair). Optimistic inaccuracy is failure to reject an incorrect program. Techniques that admit pessimistic inaccuracy but no optimistic inaccuracy are sometimes called *conservative*[3]. 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 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 technique, 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

[3]The terms "optimistic" and "pessimistic" are introduced reluctantly, there being several related terms already in common use. Unfortunately no existing pair of terms quite captures the complementary and non-exclusive nature of these two kinds of inaccuracy. The terms "accurate" and "precise" as defined in [DAW88, pg. 395] characterize closely related properties of a mapping from execution state space to model, but [You88] shows that a specification formula as well as the mapping must be considered to establish absence of optimistic inaccuracy. "Conservative" expresses both the absence of optimistic inaccuracy and the presence of pessimistic inaccuracy, but it lacks a complementary term for inaccuracy in the other direction. "Safe" is occasionally used in a sense similar to "conservative," but it is easily confounded both with freedom from hazard and with the complement of "live." "Type 1" and "Type 2" errors in hypothesis testing captures the essential idea, but which is which depends on whether the hypothesis is "the program is correct" or "the program is incorrect."

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.

Verification and testing as search. 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.

Both program proving and program testing can be characterized as searches in infinite spaces, and both must sometimes fail when an unsuccessful search is terminated. In the case of program testing, the search may terminate successfully if an error is found, while verification terminates successfully if the specification is shown to be a theorem. When either search is terminated before finding a definite answer, it is subject to inaccuracy.

	Formal verification:	*Testing:*
	Search in the (infinite) space of theorems about a program	Search in the (infinite) space of program states
Successful search:	Found theorem relating specification to program; assurance of correctness.	Found error in behavior, indicating fault in program.
Failed search:	No definite information; program *may* be incorrect. (Pessimistic)	No definite information; program *may* be correct. (Optimistic)

Of course, formal verification is not practiced as blind search through a space of theorems. In fact, the conventional decomposition of a program verification system into a verification condition generator (vcg) and theorem prover introduces an element of state-space enumeration in the verification process to greatly simplify search effort in the theorem prover. The relation between state-space analysis and formal reasoning in verification is discussed in Section 7.

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. Recently there has been some progress in the direction of verifying a stack of virtual machines (e.g., [You89b]), although such a stack necessarily has an unverified bottom level.

The alternative is to refer to a particular target machine, including translators and operating system software, on which a program is to run. This has the advantage that all levels of implementation may be validated at once, although in principle one still has to worry about the target machine wearing out. On the other hand, particular machines can have peculiarities and compensating errors that 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). Familiar examples include hardware that permits dereferencing null pointers and operating systems that zero newly allocated memory, implicitly initializing variables.

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 emphasized 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.

We adopt an operational view of a program execution as a sequence of states punctuated by operations. A precise notion of "state" and "event" will depend on a particular operational model, but informally we can think of states as memory configurations (including program counter, stack, etc.) and events as machine operations. The state space of a program is the set of states that the program can reach in all possible executions; we can think of it as a directed graph in which nodes represent program states and edges represent operations.

Folding can be formalized as a function from from graphs to graphs, usually based on a function from nodes (states) to nodes. It must be a total function (rather than a general relation) to ensure that every state is represented in the folded model. However, the the function need not be one-to-one or onto. Thus folding is more general than the related notion of *projection* [LS84], which requires every node in the simplified model to correspond to one or more nodes in the original model. Formalization of the notion of folding is carried out in more detail in [You88, You89a], but an informal understanding is sufficient for our purposes.

Often 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[4] of individual states, it is sufficient to show that each potential "bad" state in normal program execution is represented by a "bad" state in the folded model. A growing body of theory known as abstract interpretation [CC77, AH87] characterizes foldings with this property. 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 that differ only in variable values.

[4]"Safety" characterizes what a program is allowed to do, as opposed to "liveness," which characterizes what a program must do; see [Lam89] for an intuitive explanation.

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].

3.2 Sampling

A simple strategy for dealing with an infinite state space is to explore only part of it. Since the set of actual computation states of most programs is infinite,[5] all techniques normally grouped under the rubric of *dynamic analysis* sample the state space of program execution. Of course, faults may be manifested only in unexplored portion of the state space, giving rise to optimistic inaccuracy and to the old chestnut that program testing can reveal the presence of errors but not their absence.

Figure 2 suggests a simple relation between sample size (number of required test cases) and degree of optimistic inaccuracy. In reality neither axis is easy to quantify. Much research in the software testing community has been concerned with defining criteria for determining when a sufficiently representative portion of the state space has been explored to merit some confidence in the unexplored portion. Debate continues over what we should mean when we say one criterion of test adequacy is "stronger" (subject to less inaccuracy) than another [HT88, Wei89, JW89, WWH91].

It is not the purpose of this paper to provide a new way of judging the relative strengths of testing techniques (a *taxonomy* is not necessarily a *ranking*). Nevertheless we must justify the informal view of effort/accuracy tradeoffs in Figure 2 by showing that in principle it could be formalized.

The remainder of Section 3.2 makes the following arguments: Although it would be most useful to formalize accuracy of testing in terms of a statistical property like reliability or probable correctness, it is not practical to do so. It is straightforward, though, to formalize effort and accuracy in terms

[5]It is 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. If verification is not relative to particular hardware, the state space is truly infinite.

of the conventional "subsumption" and "power" relations among test adequacy criteria. This choice allows us to easily transfer existing theory about adequacy of samples drawn from a program input domain to adequacy of samples drawn from a program execution state space. Moreover, the state space framework applies equally well to deterministic and non-deterministic programs, and to testing of folded models of programs as well as conventional testing; thus transferring the subsumption hierarchy to a state-space framework increases its domain of applicability.

Statistical measures. The idea of choosing samples suggests statistical assurance of program quality. Unfortunately, even if there were agreement on what quality meant, program executions have almost no useful statistical properties from which one could infer quality. In particular, assurance based on uniform sampling of the input space (random testing) is hopeless because in general it 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 infinitesimal chance of exercising the erroneous behavior. Assuming an ideal machine with infinitely large integers, the probability of finding the fault with a finite random test set is zero, despite there being an infinite number of revealing test cases. The limitations of a real machine hardly improve the situation: Assuming 32-bit integers, the probability of a random test case revealing the fault are one in 4×10^9, despite there being 4×10^9 revealing test cases.

Reliability. A natural candidate for a quantifiable notion of program quality is reliability, measured as mean time between failures or as the probability that a single execution in actual operation will fail. For reliability estimates, a program's operational distribution of inputs is required, but usually impossible to obtain. There is no reason to believe that the projection of random inputs onto an operational distribution is any more uniform than their projection onto the complete state space of program execution.

Lipton [Lip89] has proposed an approach that treats the operational distribution as an adversary function. Lipton's method introduces randomness in a way that replaces inputs chosen by the adversary by small sets of inputs uniformly distributed in the input space, making it possible to draw valid statistical conclusions about reliability from random testing. Such a testing scheme is called "distribution free," because unlike conventional techniques for measuring reliability it does not depend on any properties of the true operational distribution of inputs. Unfortunately the class of systems for which distribution free testing schemes are known is small, and it does not appear easy to find distribution-free testing schemes for common non-mathematical functions.

Probable correctness. Instead of confidence in reliability, one might seek statistical assurance that a program is completely free of faults. Hamlet pointed out in [Ham87] that 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 than to the space of input data. The results of probable correctness theory so far are mostly negative; [HT88] shows that partitioning (which includes all functional and structural test 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, since one cannot in general assume anything about the distribution of samples drawn from the input space.

In the absence of a sound analytical basis for inferring a statistical measure of program quality from testing, one might hope to infer reliability or some other statistical measure on the basis of empirical evidence of efficacy of particular testing methods. This remains a hope for the future, but the body of empirical evidence is not yet available. We are left therefore with non-statistical characterizations of effort/accuracy tradeoffs in testing. The "subsumption" and "power" relations, despite their flaws, are well understood and easy to transfer to our state-space framework.

The subsumption hierarchy. An adequacy criterion accepts or rejects a sample of program behaviors as an "adequate" test set, sometimes depending on external information such as the program specification or program text. The most thoroughly studied means for comparing software test adequacy criteria is the "subsumption" hierarchy. Criterion A subsumes criterion B iff, for all programs P and specifications S, any test set that satisfies A for

P and S also satisfies B for the same P and S. Gourlay has formalized the subsumption relation [Gou83] and shown that if criterion A subsumes B, then for any program P and specification S, testing P to criterion A can admit optimistic inaccuracy only if testing P to criterion B also admits optimistic inaccuracy. The latter characterization of optimistic inaccuracy is sometimes called the "power" relation. Thus the subsumption hierarchy orders test adequacy criteria both by minimum required sample size and by degree of optimistic inaccuracy, and is a reasonable if imperfect characterization of cost/effort tradeoffs in sampling.

Since the subsumption hierarchy was developed to compare conventional testing techniques for deterministic sequential programs, it is defined as a relation involving sets of test data. An adequacy criterion for test data induces an equivalent criterion for coverage of an execution state space. The converse is not true for programs with non-determinism, including concurrent programs.

A test adequacy criterion for test data sets can be defined (as in [Wey86] and [Wei89]) as a relation among programs, specifications, and sets of input vectors. It is convenient to write such a relation as

$$A\colon S \times P \rightarrow (A_I\colon 2^I \rightarrow \text{boolean})$$

where S ranges over specifications, P ranges over programs, and I ranges over input vectors (hence 2^I ranges over test data sets). If, as is usual, we consider unexpended input as part of a program state, then it is trivial to define an A_H that mimics any A_I (because an input vector uniquely determines an initial state). The reverse is also true for deterministic execution, because input vectors are in one-to-one correspondence with paths through a state space.

In the case of non-determinism (e.g., scheduling decisions in a concurrent program), several possible paths through the state space may correspond to a single input vector. Given any adequacy criterion A_I for test data, it remains trivial to define an equivalent criterion A_H; but given an A_H, there may be no equivalent function A_I. The adequacy criteria described in [TK86, TKL89], for instance, do not correspond to any relation on programs, specifications, and test data sets alone.

Unfortunately, and contrary to what Figure 2 may suggest, subsumption is only a partial order and many common adequacy criteria are incomparable with respect to subsumption [CPRZ85]. Moreover, the common criteria in Figure 2 (statement testing, branch testing, etc.) deal inadequately with undecidability problems in testing. A criterion is trivially free of optimistic inaccuracy but also useless when it requires execution of an unexecutable path; the relation between subsumption and degree of inaccuracy also breaks down when considering such “inapplicable” criteria [Wei89]. On the other hand, redefining the criteria to require exercising only executable paths makes satisfaction of the criteria undecidable. (For mutation analysis and related techniques, the analogous problem is recognizing equivalent mutants.) Frankl and Weyuker [FW88] show that excusing unexecutable paths from adequacy criteria changes some relations in the subsumption hierarchy. Determining whether a path is executable, or finding test data to exercise a particular path, could in principle require as much effort as formal verification, so equating sample size with testing effort is a considerable oversimplification of true effort/accuracy tradeoffs.

Partition testing. Many test adequacy criteria attempt to force selection of a representative sample of the execution state space by partitioning behaviors into classes, and requiring samples to be drawn from each class. Several fault-based and error-based criteria for selecting test data have been put forward. Mutation testing [DLS78] is most direct: A sample of program behaviors is judged adequate when it differentiates between the actual program and a set of variant programs with seeded 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. 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 [ROT89, OB88, RC85].

Control flow and data flow coverage criteria [RW82] 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. For instance, branch testing requires that every branch in a program text be executed at least once; each branch is executed in some subset of possible program executions, and the sample of the state space is not considered adequate unless it includes at least one behavior from each subset. Although the grouping of behaviors can be related to a folded model of the state space (control flow graphs in the case of branch testing), actual program executions rather than abstract models of execution are tested. Thus we consider such criteria as constraints on drawing samples rather than as analysis of folded models.

Sampling folded models. 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]. Similar techniques are employed in analysis of executable specification formalisms including StateCharts [HLN+90] and Paisley [ZS86]. These techniques have not been treated in the same literature as conventional software testing, but they present similar effort/accuracy tradeoffs.

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 that 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 testing* and *global symbolic execution*. 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. The interested reader can find thorough introductions to theoretical and practical aspects of symbolic evaluation in [HK76] and [CR81], respectively. A modern symbolic execution system able to carry out either of the two techniques described below is presented in [KE85].

4.1 Symbolic testing

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 that 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 non-deterministic choice were possible, the current token location would also be required.) For a program without loops, this state space is a finite 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 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 many ways. It admits the same sort of optimistic inaccuracy (since faults may lie on paths that are not explored), requires an oracle ("Is the final path expression acceptable?"), and like conventional program testing may be practiced to some coverage criterion. The main difference is that a single symbolic execution usually represents a large number of actual executions; but a symbolic execution may also represent zero actual executions, since it is impossible in general to determine whether the path followed is executable.

4.2 Global symbolic execution

Whereas symbolic testing explores a sample of the state space, global symbolic execution 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 execution is like other static analysis techniques.

Global symbolic execution folds together states reached along paths that differ only in the number of iterations a loop is traversed. An approach known as global symbolic evaluation represents the effect of a loop by a set of recurrence relations, which in some cases can be simplified to closed form expressions [CHT79, CR81]. An alternative, especially suitable when symbolic evaluation is used as an aid to formal verification, is to cut each loop with an assertion. Since recurrence relations and closed form solutions produced in global symbolic evaluation can also be represented as assertions, the difference between the techniques is in division of labor between the programmer and the symbolic evaluation system. Global symbolic execution with user-supplied assertions is the approach considered here, for simplicity and because, fifteen years after their introduction, symbolic evaluation techniques without a large measure of user guidance do not seem promising.

Imagine that every loop in a program is cut by a loop invariant assertion that 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 that prevent too much information from passing through.[6]

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, and ignoring again differences between ideal and real machines). Pessimistic inaccuracy results primarily from the difficulty of finding satisfactory loop-cutting assertions (although failure of a simplifier or theorem prover may introduce additional pessimistic inaccuracy). 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.

[6]An important advantage of user-supplied assertions over the global symbolic evaluation approach is that the coarseness of the folding can be controlled; whereas the global symbolic evaluation must attempt an exact characterization of a program function, users may choose to prove weaker assertions.

4.3 Summary

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

Oracles. Among techniques that 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.

In common use a test oracle is a decision procedure for accepting or rejecting a test outcome. When a program or unit under test is intended to compute a function of its inputs, a test oracle compares (input, output) pairs. It is useful to adopt a somewhat broader view of oracles to encompass other means of accepting or rejecting behaviors.

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)? Checking (input,output) pairs is a special case of a state oracle that examines only initial and final states (we can consider input data and output data to be parts of the state). Assertion checking systems like the Anna toolset [LvH85] check intermediate states of a computation. Path-oriented oracles must either accumulate "historical" information in addition to the program state (like TSL [HL85, Ros91]) or else operate on an explicit representation of the state space (e.g., an execution trace or a directed graph representation of reachable 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, classical testing, and folded 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 and complete 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 or summarized into a finite set of representative values. *Sampling:* None.

Model schema: The model schema is a directed graph that can itself be considered as a state space, and the analysis procedure annotates 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 are suitable for use as oracles when combined with other techniques for constructing a representation of the state space.

Representation of the state space: Explicit, given as input. Conventional data flow analysis starts from an augmented control flow graph, but flow analysis techniques can also be applied to other directed graph models of program behavior, including those produced by reachability analysis (see below).

Oracle: Hard-wired classes of anomalous behavior in the case of systems like DAVE [OF76], programmable and path-oriented in [OO86] and [CES86].

Reachability analysis.

Folding: Control flow is modeled, data values are ignored or summarized. Only bounded variables (e.g., booleans and small integers) can be fully modeled.

Sampling: None. In common use, the term "reachability analysis" refers to an exhaustive enumeration of reachable states. Non-exhaustive techniques using the same model schemata are classified as *folded testing,* below.

Model schema: Various graph models including Petri nets and program control flow graphs.

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 hardwired 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. A reachability analysis technique for constructing a representation of a state space is typically combined with a flow analysis technique serving as oracle.

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 specification-based testing).

A coverage criterion may be based on an folded auxiliary model of execution. For instance, data flow testing [RW82, CPRZ85] measures representativeness in terms of coverage of certain control-flow subpaths, thereby relating program execution to paths in a flowgraph model. We consider such a model auxiliary because it serves only to control sampling; completely detailed program states are generated during execution, and a test oracle judges actual program states rather than folded states.

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. When a coverage criterion is applied, an additional record is accumulated to indicate which portions of the state space have been explored.

Oracle: Testing techniques focusing on exploration of the state space, such as structural and functional coverage schemes, usually leave the oracle unspecified. (An exception is Howden's work on testing mathematical functions [How78], in which the adequacy criterion explicitly concerns selecting (input,output) pairs on which to compare a program function with its specification.) Assertion-checking schemes (e.g. Anna [LvH85]) are programmable state-oriented oracles; TSL [LHM$^+$87] is programmable and path-oriented. There is some room for mix-and-match between coverage criteria and oracles, as there is between reachability analysis techniques and flow analysis techniques.

Folded testing.

Folding: Classes of data are folded in the case of symbolic testing. Additional folding of implementation details may occur in simulations of executable specifications such as Petri nets or PAISley [ZS86]. (In this use, an executable "specification" takes the role of an implementation, and is tested against some higher-level specification.) In folded testing techniques, the state space is considerably simplified in comparison to actual program execution, but may still be too large to enumerate exhaustively.

A folded testing technique may be used in place of a reachability analysis technique, using the same folding and model schema as complete reachability analysis, if the reachability graph becomes too large to enumerate exhaustively. These techniques often go by other names: "simulation" for exploration of individual, randomly chosen paths without backtracking, or "scatter search" for non-exhaustive state space exploration with limited backtracking [Hol87].

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 it is 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 analysis [Hol87] is an example of folded testing applied in place of exhaustive reachability analysis; Argos provides for heuristic guidance to a "scatter search" of the state space.

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 and other objects 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. Explicit consideration of how each folds and/or samples the state space of program execution is helpful in considering how they can be combined. Section 7 continues in this vein by considering how a framework based on state-space analysis can be useful in devising hybrid techniques involving formal verification.

6.1 Combining attributes: One from column A, one from column B

Many analysis techniques prescribe some attributes of the classification scheme described here, and leave others unresolved. One may treat the attributes as a sort of check list for putting together a complete technique. That is, a technique has not been completely specified until each attribute (folding, sampling, schemata, representation, and oracle) has been determined.

For instance, one can imagine a symbolic testing technique, using the Anna specification language and processor [LvH85] to provide oracles, and testing to an "all-def-use-paths" data flow coverage criterion. Each of these techniques individually leaves several attributes unspecified, but together they determine each of the attributes. Symbolic testing determines the model schemata, state folding, and representation of the state space, while the data flow coverage criterion determines the degree of sampling. Instrumentation of a program by the Anna processor provides oracles for a class of specifications. One might adjust any one of these parameters independently, for instance using conventional testing in place of symbolic testing (changing the degree of folding) or using a specification-based or fault-based test adequacy criterion in place of data flow coverage.

6.2 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 that 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 that uses pessimistic techniques to concentrate optimistic techniques has been described in detail by Osterweil [Ost84]. A combination of a pessimistic verification technique and a state-space exploration technique in which information from a failed verification is used to guide sampling is described in Section 7.3.

6.3 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 generally a more pessimistic approach than interpreting the time information. 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. The standard interpretation will generate a sequence of firings for every sequence of firings in the time interpretation. This will make it pessimistic for most classes of errors, although it is possible to compose a branching time temporal logic assertion that may be satisfied only by the standard interpretation, e.g., an assertion that a certain state is reachable but not inevitable. The rules in [You88] can be used to prove that the standard interpretation preserves violations of a particular temporal specification formula.

6.4 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 that 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. Of course the representativeness of samples is never guaranteed; we may choose to divide the state space in more than one way in hope that some classes or combination of classes will concentrate faulty behaviors as we require.

Neither specification-based (functional) nor code-based (structural) test adequacy criteria are good at uncovering all kinds of faults. Code-based criteria are generally blind to missing cases, such as a feature or condition in a specification that has been neglected in the implementation. Specification-based criteria are not sensitive to aspects of behavior that are not manifested externally, such as collisions in a hash table or rebalancing of an AVL tree. It is natural, therefore, to consider whether some combination of specification-based and code-based test adequacy criteria can combine their advantages.

One proposal for combining specification- and code-based adequacy criteria is to intersect their respective partitions [RC85]. However, the proposition that the smaller partitions thus formed are advantageous has been disputed [Ham87, Ham89]. Whatever the value of intersecting partitions, it will certainly be expensive to form the intersection and find test data to meet the criterion.

A simple-minded strategy for combining criteria, which may nevertheless be effective, is to consider a test set adequate when it satisfies each of a set of different and otherwise reasonable criteria. This corresponds to the view that the goal of a test adequacy criterion is not to find demonstrably good tests, but to reject demonstrably inadequate sets of test cases.

7 Proving and testing

The classification scheme described above focuses primarily on techniques that in some way explore the state space of program execution. Since a purpose of the revised taxonomy is to suggest synergistic combinations of techniques, it is worthwhile to consider how these state-space techniques are related to others. In particular, how are state-space analysis techniques including testing related to formal verification techniques?

In Section 3 formal verification was described abstractly as search for a specification formula in the space of theorems about a program, and contrasted to search in the state space of program execution. On the other hand, global symbolic execution (Section 4.2) is the classical Floyd/Hoare style of verification with some mechanical help. Formal verification techniques often combine enumeration of folded program states with a formal reasoning technique, with the formal reasoning component (e.g., theorem prover) filling the role of oracle. Consideration of the folding used in state exploration can be a basis for understanding combinations of testing with formal verification techniques and for devising new combinations.

In the remainder of this section, we first elaborate somewhat on the view of formal verification as folded state-space exploration. Then two hybrid techniques from the literature are examined in this light. Kieburtz and Silberschatz' approach to verifying "access-right expressions" is an example of a combination technique that combines flow analysis, manual formal verification, and selective run-time checking. The constrained expressions analysis approach of Avrunin, Dillon, and Wileden is an illustrative hybrid technique that first attempts to verify critical properties with a minimum of state-space exploration, and failing that uses information from the formal reasoning component to constrain search in a folded state-space model (a combination of folding and sampling).

7.1 Folding in program verification

A formal verification system is typically composed of a verification condition generator (vcg) and a theorem prover. A global symbolic evaluator of the kind described in Section 4.2 can serve as a verification condition generator [KE85], which as we have observed folds the execution state space into a vastly smaller number of states associated with intermediate assertions.

The theorem prover searches in the space of theorems, but the vcg combines information about intended behavior (the specification) with information about actual behavior (the program) in a way that transforms the blind search into a series of simpler goal-directed searches. Axioms describing program behavior are applied only in the vcg, where they are combined with guidance in the form of conjectures (intermediate assertions) to produce a collection of logical formulae to be verified by the theorem prover. Showing that each logical formula is a theorem involves search, but the searches are independent.

Two kinds of state-space folding can be identified in formal verification. First, intermediate assertions define a mapping from actual program states to abstract states described by assertions; this is the folding described in Section 4.2. Second, verification normally proceeds in a series of levels of abstraction. The hierarchical structure of abstractions necessary for practical verification is mirrored by a set of foldings of the execution state space.

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 choosing the right abstractions, i.e., determining how to fold the space to make the next inference step easy. In tool-supported verification, this creative step is left to the programmer.

7.2 Access-right expressions

Several researchers have pointed out that verification techniques for concurrent systems involve tradeoffs between amount of state enumeration (degree of folding) and complexity of formal reasoning. A typical problem involves verifying the behavior of a data abstraction whose operations may be invoked by multiple processes. In the classical method for verifying a data structure protected by a monitor, an invariant assertion folds the state space into a single state, with a self-loop required to maintain the invariant. The access-right expression approach to specifying and verifying monitors is an example of a technique that performs slightly more state-space enumeration (less folding) to simplify the invariants associated with each state.

Kieburtz and Silberschatz proposed in [KS83] a method of specifying and enforcing sequencing restrictions in concurrent programs, and verifying monitor invariants subject to those sequencing restrictions. Their enforcement strategy is a hybrid of formal verification, flow analysis, and run-time monitoring. They associate a set of *access-right expressions* with each resource class. A resource class is essentially an abstract data type managed by a monitor. An access-right expression is a regular expression describing a legal sequence of operations. A single resource class may be associated with more than one access-right expression, each describing a particular class of service (e.g., "reader" access vs. "writer" access in a mutual exclusion problem).

Kieburtz and Silberschatz observe that the specification of a resource is often conditional upon correct use of the resource, i.e., the specification is not "Invariant I is maintained" but rather "Invariant I is maintained provided each client obeys protocol P." The purpose of access-right expressions is to formally specify a usage protocol in a form that can be verified in each client and assumed in the verification of the resource monitor.

Verification of a resource with access-right expressions involves showing external consistency (each client obeys the protocol) and internal consistency (the monitor behaves correctly, assuming the protocol is obeyed). External consistency is enforced through a combination of flow analysis and run-time monitoring. First, flow analysis is applied in an attempt to verify that every path in the control flow graph of a client obeys the sequencing constraint. (Olender and Osterweil have devised and implemented procedures for performing such a check [OO86, OO89].) If flow analysis fails to verify correct sequencing, the constraint is represented as an automaton and monitored at run-time.

The control flow graph used in flow analysis is a folded model of actual execution. When exhaustive analysis of the folded model fails to verify correct sequencing, it could be because of pessimistic inaccuracy introduced in the folding. Run-time monitoring is free of this pessimistic inaccuracy, but subject to optimistic inaccuracy in the sense that monitoring a sample of the execution state-space without encountering sequence violations does not justify an inference that future executions will also be free of violations.

Verification of the monitor involves replacing the usual monitor invariant assertion by a set of assertions describing the state of the resource controlled by a monitor after particular operation sequences. That is, each access-right expression is represented by a finite state automaton, each state Q is associated with an assertion I_Q which at least implies the global invariant, and each edge is associated with an operation P managed by the monitor. The global (and weakest) invariant is associated with the initial state to facilitate induction over the number of client processes. A monitor meets its obligation with respect to a single client if $I_Q\{O\}I_R$ for each edge $Q \xrightarrow{O} R$ in the automaton. Non-interference among an arbitrary number of clients is established by showing that each I_Q is maintained by every non-repeating path prefix of each automaton.

7.3 Constrained expression analysis

The above discussion of techniques combining formal verification and state space analysis have treated the formal reasoning (theorem prover) component of a formal verification system essentially as a black box in the role of oracle function. In mixed verification strategies, one would like to take advantage of the formal reasoning component even when it "fails" in the sense of not proving an assertion. If search in the space of theorems about a program fails, it may still produce information that can be used to control search in the space of execution states.

Constrained expression analysis [ADWR86, DAW88, ADW89] is an example of a technique with aspects of both formal reasoning and state-space analysis. The approach to constrained expression analysis described in [ADW89] involves an attempt at verification followed (when verification fails) by search for an erroneous behavior. Information produced in the failed verification step is used to guide the state-space exploration step.

Constrained expressions are a formalism for describing sequences of interactions among communicating processes. Like many such formalisms, they are based on regular expressions with extensions to express concurrent activity as interleaving. A specification of *intended* behavior (prohibited behavior, rather) is expressed in the same constrained expression formalism as the description of *actual* behavior. Let L_P be the language generated by the constrained expression representation of prohibited behavior, and let L_A be the language generated by the constrained expression representation of actual behavior, projected onto the alphabet of L_P. A set of inequalities characterizing strings in $L_A \cap L_P$ is generated and submitted to an inequality solver. If the system of inequalities is inconsistent, then $L_A \cap L_P = \emptyset$, meaning the system exhibits only correct sequences of events.

Although the constrained expression representation of a system need not have any pessimistic or optimistic inaccuracy,[7] inequalities derived from the constrained expressions do not completely characterize possible event orderings. Pessimistic inaccuracy is introduced in deriving inequalities from constrained expressions: the set of inequalities may have a solution, even though the behavior they (partially) describe is impossible. One could continue to generate inequalities describing a set of constrained expressions in more and more detail, as one could in principle endlessly enumerate theorems about a program. (The inequalities are in fact theorems about behaviors of the system, albeit in an unusual form.)

Instead of generating more and more inequalities, the tools described in [ADW89] fall back on a typical state-space exploration strategy. A "behavior generator" generates example behaviors from a constrained expression representation. This step is similar to reachability analysis or folded testing, but search in the space of actual behavior is guided by information gained in the prior analysis. A solution to inequalities generated from constrained expression analysis is used by the behavior generator in an attempt to find and exhibit a prohibited behavior.

Constrained expression analysis is actually a hybrid of a pessimistic formal verification technique (inequality generation) and an optimistic state-

[7][DAW88] shows that constrained expressions are sufficiently powerful to represent some other design notations (including Petri nets) without inaccuracy. If these design notations are considered folded models of programs, then constrained expressions derived from design notations inherit any inaccuracy incurred in folding the original program.

space technique with sampling (behavior generation). From the perspective of our taxonomy, the most interesting aspect of constrained expression analysis is the synergism achieved by combining them. The general principle, which may be applicable to other combinations of pessimistic and optimistic techniques, is using information characterizing all prohibited behaviors to guide sampling by the optimistic technique.

8 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 that limit effort by folding states together) and *optimistic* inaccuracy (characteristic of techniques that 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. It is nonsense for a technique to be both "static" and "dynamic," but folding and sampling can be applied together as well as separately. The revised taxonomy is thus better able to accommodate techniques that combine folding and sampling, including non-exhaustive variants of reachability analysis and symbolic testing.

Any classification scheme highlights some attributes of the things to be classified and deemphasizes others. A taxonomy must be judged by whether the distinctions it highlights and the organization it imposes are useful. The folding/sampling distinction highlights design tradeoffs inherent in devising fault detection techniques, and the organization it imposes facilitates consideration of potential synergism among techniques.

References

[ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, June 1982.

[ADW89] George S. Avrunin, Laura K. Dillon, and Jack C. Wileden. Experiments with automated constrained expression analysis of concurrent software systems. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 124–130, Key West, Florida, December 1989. ACM SIGSOFT. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).

[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 Transactions on Software Engineering*, SE-12(2):278–292, February 1986.

[AH87] Samson Abramsky and Chris Hankin. *Abstract Interpretation of Declarative Languages*. Halstead press, New York, 1987.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

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

[CHT79] Thomas E. Cheatham, Jr., Glen H. Holloway, and Judy A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, SE–5(4):402–417, July 1979.

[CPRZ85] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244–251, London, August 1985. ACM SIGSOFT.

[CR81] Lori A. Clarke and Debra J. Richardson. Symbolic evaluation methods — implementations and applications. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 65–102. North-Holland, 1981.

[DAW88] Laura K. Dillon, George S. Avrunin, and Jack C. Wileden. Constrained expressions: Toward broad applicability of analysis methods for distributed software systems. *ACM Transactions on Programming Languages and Systems*, 10(3):374–402, July 1988.

[DLS78] Richard DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4), April 1978.

[EFRV86] Gerald Estrin, Robert S. Fenchel, Rami R. Razouk, and Mary K. Vernon. SARA (System ARchitects Apprentice): Modeling, analysis, and simulation suppor for design of concurrent systems. *IEEE Transactions on Software Engineering*, 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 *Proceedings of the 5th International Workshop on Protocol Specification, Testing, and Verification*, Toulouse, France, June 1985.

[FW88] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[Gou83] John S. Gourlay. A Mathematical Framework for the Investigation of Testing. *IEEE Transactions on Software Engineering*, SE-9(6):686–709, November 1983.

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

[Ham89] Richard Hamlet. Theoretical comparison of testing methods. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 28–37, Key West, Florida, December 1989. ACM SIGSOFT. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).

[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.

[HL85] David Helmbold and David Luckham. TSL: Task sequencing language. In John G. P. Barnes and Gerald A. Fisher, Jr., editors, *Ada in Use: Proceedings of the Ada International Conference*, volume 5, pages 255–274, Paris, May 1985. Association for Computing Machinery and Ada-Europe, Cambridge University Press.

[HLN$^+$90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

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

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

[How78] W. E. Howden. Algebraic program testing. *Acta Informatica*, 10, 1978.

[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 *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 206–215, Banff, Canada, July 1988. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[JW89] Bingchiang Jeng and Elaine J. Weyuker. Some observations on partition testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 38–47, Key West, Florida, December 1989. ACM SIGSOFT. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).

[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.

[KS83] Richard B. Kieburtz and Abraham Silberschatz. Access-right expressions. *ACM Transactions on Programming Languages and Systems*, 5(1):78–96, January 1983.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989. Also appeared as Technical Report 15, DEC Systems Research Center.

[LC89] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the Eleventh International Conference on Software Engineering*, 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.

[Lip89] Richard J. Lipton. New directions in testing. Technical report, Department of Computer Science, Princeton University, 1989.

[LS84] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.

[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, Department of Information and Computer Science, University of 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 Transactions on Software Engineering*, SE–13(10):1080–1091, October 1987.

[OB88] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[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 *Proceedings of the Workshop on Software Testing*, pages 14–22, Banff, Canada, July 1986. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[OO89] Kurt M. Olender and Leon J. Osterweil. Cesar: A static sequencing constraint analyzer. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 66–74, Key West, Florida, December 1989. ACM SIGSOFT. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).

[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, University of California, 1987.

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

[Ros91] David S. Rosenblum. Specifying concurrent systems with TSL. *IEEE Software*, 8(3):52–61, May 1991.

[ROT89] Debra J. Richardson, Owen O'Malley, and Cindy Tittle. Approaches to specification-based testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 86–96, Key West, Florida, December 1989. ACM SIGSOFT. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).

[RW82] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of the Sixth International Conference on Software Engineering*, pages 272–278, Tokyo, Japan, September 1982.

[Tay83] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the 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.

[TK86] Richard N. Taylor and Cheryl D. Kelly. Structural testing of concurrent programs. In *Proceedings of the Workshop on Software Testing*, pages 164–169, Banff, Canada, July 1986. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[TKL89] Richard N. Taylor, Cheryl D. Kelly, and David L. Levine. Structural testing of concurrent programs. Arcadia Technical Report UCI-89-22, University of California, Irvine, October 1989.

[Wei89] Stewart N. Weiss. What to compare when comparing test data adequacy criteria. *ACM SIGSOFT Software Engineering Notes*, 14(6):42–49, October 1989.

[Wey86] Elaine J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–1138, December 1986.

[WWH91] Elaine J. Weyuker, Stewart N. Weiss, and Dick Hamlet. Comparison of program testing strategies. In *Proceedings of the 4th ACM Symposium on Software Testing, Analysis, and Verification*, Victoria, British Columbia, October 1991.

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

[You89a] Michal Young. *Hybrid Analysis Techniques for Software Fault Detection*. PhD thesis, University of California, Irvine, 1989. Available as UCI ICS technical report 89–26.

[You89b] William D. Young. Verified compilation in micro-gypsy. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 20–26, Key West, Florida, December 1989. ACM SIGSOFT. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).

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

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