

Unifying Verification and Validation Techniques : Relating Behavior and Properties through Partial Evidence*

Matthew B. Dwyer, Sebastian Elbaum
Dept. of Computer Science and Engineering
University of Nebraska – Lincoln
{dwyer,elbaum}@cse.unl.edu

ABSTRACT

The past decade has produced a range of techniques for assessing the correctness of software systems. These techniques, such as various forms of static analysis, automated verification, and test generation, are capable of producing a variety of forms of evidence showing that the software behavior meets its specified properties. We contend that, as currently formulated, existing techniques fail to externalize all of the useful pieces of evidence that they compute which limits the opportunities to obtain a comprehensive and accurate assessment of property-behavior conformance. Explicitly accounting for the ways that V&V techniques produce *partial evidence* offers the potential to look beyond the boundaries of individual analysis, verification, and testing techniques to consider the larger question of how the techniques fit together to provide an explicit body of evidence about software system quality.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Formal methods, Validation*; D.2.5 [Software Engineering]: Testing and Debugging—*Symbolic execution, Testing tools*

General Terms

Reliability, Verification

1. INTRODUCTION

The past decade has seen the development of a large number of techniques for reasoning about software system correctness. For the sake of illustration, consider work on (1) software model checking, e.g., [2, 19], (2) SAT-based software verification, e.g., [3, 17],

*This material is based in part upon work supported by the National Aeronautics and Space Administration under grant number NNX08AV20A, by the National Science Foundation under awards CNS-0720654 and CCF-0915526, and by the Air Force Office of Scientific Research under Award #9550-09-1-0687. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NASA, AFOSR, or NSF.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11 ...\$10.00.

(3) advanced abstract interpretation, e.g., [4], (4) symbolic execution, e.g., [16, 20], and (5) runtime verification, e.g., [1, 6]. In each of these areas techniques have matured, through the accumulation of a series of technical advances made by a number of different researchers, to the point where they can be applied to components of real software systems to detect faults or to assure limited correctness properties.

Many of these areas remain very active and are likely to continue to be “hot” for many more years. We contend, however, that the communities of experts who have developed and matured these techniques risk missing opportunities to advance the overall goal of software system V&V by narrowly focusing on individual techniques. As Oliver suggests, in “The Incomplete Guide to the Art of Discovery” [12], it can be healthy to disrupt the standard “flow” in a discipline by raising fundamental questions about the utility and broader connections of the specific methods developed in a research program.

We believe that the recent NRC study “Software for Dependable Systems : Sufficient Evidence?” [10] provides a valuable starting point to frame those broader questions to disrupt the flow of ideas in the verification and validation community. The report makes a series of recommendations including “Make a dependability case for a given system and context: evidence, explicitness, and expertise.” Here we wish to expand on the meaning of “evidence”. The study’s authors suggest that evidence will take the form of a “dependability case”. We will not expand on the full definition of a dependability case here, but will note that the report recommends that a “case will typically combine evidence from testing with evidence from analysis”. Implicit in this recommendation is the need to represent evidence of property satisfaction that is generated from multiple, and potentially very different, V&V techniques. This is the key broader challenge we seek to expose.

We begin by considering the type of evidence that might be produced by an “ideal” analysis¹ or testing technique. An ideal analysis targets precisely specified properties, uses a sound abstraction of the software’s semantics² and seeks to produce a proof that all program executions satisfy the property³. An ideal testing process uses a test oracle that directly encodes (a set of) properties to be checked, and selects a diverse yet minimal set of inputs that, when applied under specified controlled conditions, force the program to execute

¹Here we use the term *analysis* to mean any technique intended to reason about all program executions; for instance, static analyses such as [4] or software verification techniques, such as [2].

²Rather than formalizing soundness here, we simply appeal to the intuition that a sound abstraction overestimates the set of feasible program executions.

³For simplicity we restrict our discussion to the common case where properties describe conditions that are intended to hold on all program executions.

in ways that are representative of all of its different behaviors. An ideal analysis and testing technique also provides additional supporting data that helps to contextualize the computed results and to serve as building blocks on which other techniques can build.

These ideals are just that. In reality, testing and analysis techniques render much weaker forms of evidence, *partial evidence*. Nearly all static analysis tools are unsound in their treatment of real programs, for example, in their use of integer arithmetic rather than bit-limited computer arithmetic, and in their treatment of certain exceptions such as Java’s `OutOfMemoryError`. Moreover, it is extremely rare for a static analysis to scale to large programs and be sufficiently precise to prove property conformance. It is much more common for an analysis to produce a, hopefully small, set of potential property violation reports for a developer to inspect [9]. Similarly, state-of-the-practice testing uses test oracles derived from informal descriptions of system requirements and seeks input sets that achieve only the simplest forms of coverage of the program’s syntax, e.g., statement or branch coverage. The former weakness can be mitigated in part by leveraging explicitly defined correctness properties when available, while the latter requires that testing moves more towards stronger oracles and execution path-based adequacy criteria. Still, both mitigation strategies pose serious scalability challenges.

This situation is aggravated as the techniques discard some of the computed and potentially valuable partial evidence. This runs directly opposite to another relevant recommendation from the NRC study [10] which states “Demand more transparency, so that customers and users can make more informed judgments about dependability.” The results of applying a technique should be rendered in a form that makes it more visible to the stakeholders in a system, including other techniques. Existing state-of-the-art analysis and verification techniques provide an *all or nothing* interface in rendering their results – either a property is proved or a series of reports of potential errors is issued. As we explain, this limits opportunities for revealing what the application of a technique to a system has discovered and how that information might be integrated with results from applying other techniques.

In this work we start framing the challenge of integrating multiple forms of partial evidence by relating behavioral coverage with program properties.

2. RELATING SYSTEM BEHAVIOR AND PROPERTIES

Intuitively, a system’s *behavior* is defined as the set, or sequence, of output values computed in response to a given set, or sequence, of input values. An alternative, but equivalent, characterization can be given in terms of the sequence of statements, or branches, executed in response to a given set of input values.

V&V techniques make judgments about system behavior by checking *properties* of output values and their relationships to input values. For a given system there may be (infinitely) many behaviors and many properties; Figure 1 depicts this relation. An ideal V&V process will consider each behavior-property pair, for example the one marked with the “x”.

Testing, and dynamic analyses in general, focus on a single behavior at a time, using input values to drive system execution and checking the resulting output values. The thin gray vertical line on the left of Figure 1 depicts a single test case equipped with an oracle for each property p_1, \dots, p_n ; the solid black squares indicate property checks that are encoded in test oracles. More typically, testing will only focus on a subset of a system’s properties. The

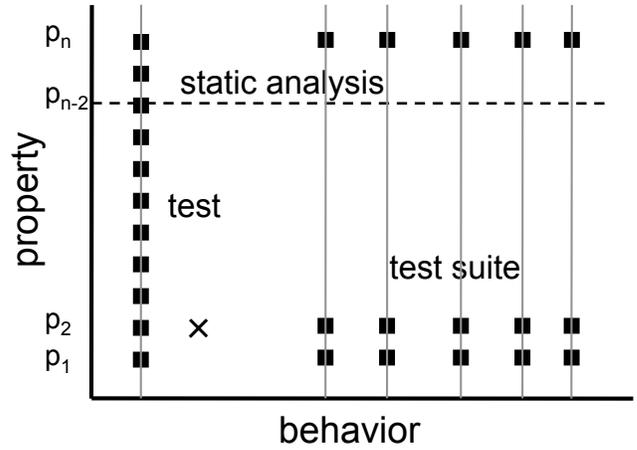


Figure 1: Relating Behaviors and Properties

test suite depicted on the right of the Figure include oracles for properties p_1, p_2 , and p_n .

In contrast to testing, a sound static analysis typically focuses on a single property at a time and attempts to reason about all possible program behaviors. The horizontal dashed line in the Figure depicts such an analysis for p_{n-2} . This depiction corresponds to a static analysis that is able to conclusively demonstrate that p_{n-2} holds on all behaviors of the system. In practice, it is often the case that a static analysis will be unable to produce such a conclusive result [14], perhaps because the analysis is imprecise, and instead will issue reports of potentially erroneous system behaviors. Note that these reports may be “false” in the sense that no actual error exists.

Independent of whether the approach is static or dynamic, given the size of the system behavior and property relation, one cannot expect any one technique to provide significant property-behavior coverage. However, as a series of V&V activities are performed a body of evidence, i.e., the property-behavior pairs in the Figure, accumulates. To leverage this a key question must be addressed: “What property-behavior coverage is actually provided by applying a V&V technique to a software system?”.

3. PARTIAL EVIDENCE

Figure 1 is an inaccurate depiction of what occurs in practice in (at least) three ways. First, most applications of static analysis techniques to real systems result in only partial behavior coverage. Second, when running a test, or a dynamic analysis technique, information about a set of system behaviors can be inferred even though a single behavior is considered explicitly. Finally, many V&V techniques do not consider the original system correctness requirements directly, rather they target a set of derivative, and perhaps simplified, properties.

We believe that explicitly accounting for the ways that V&V techniques produce *partial evidence* offers the potential for a more accurate and comprehensive characterization of property-behavior coverage. We illustrate three different ways that partial evidence arises in the remainder of this section.

Figure 2 focuses on a single property to illustrate these issues (the Figure depicts an exploded view of a single point on the y-axis of Figure 1). The x-axis, at the bottom of the Figure, illustrates the set of all system behaviors.

The first scenario we consider is *extracting partial evidence* from an analysis that aims for a conclusive determination of property

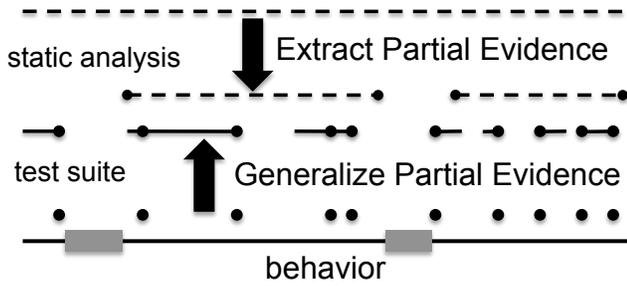


Figure 2: The lines depict the behavior coverage provided by a static analyses and test suite for a single program property

conformance. As explained above, when a sound static analysis (the dashed line at the top of the Figure 2) is performed, it aims to produce a conclusive demonstration that the property holds and if that fails it produces a set of error reports.

Such an analysis may fail because some system behavior does not conform to the property, but it is often the case that for real systems a conclusive result is not obtained due to imprecision in the analysis. Imprecision arises when analysis designers attempt to control analysis cost by enforcing an abstraction of system execution semantics. As we have observed in our own work [8], it is often the case that even a very imprecise analysis is able to demonstrate that some subset of system behaviors enjoys the property under analysis. When those behaviors can be explicitly characterized, then they too can contribute to the body of V&V evidence.

When a static analysis does produce a conclusive result, there may be questions about how thoroughly it accounted for all system behaviors. One reason for this is that even sophisticated static analysis implementations, e.g., [18], may admit unsound treatment of specific language features for performance or scalability reasons, e.g., dynamic class loading, calls to native or foreign language routines. This may mean that a demonstration of property-behavior conformance provides only partial information – a property may be violated on a behavior that is treated unsoundly. Unsoundness may be incorporated into an analysis intentionally to boost performance while aiming just for fault detection, e.g., [5].

Regardless of the source of unsoundness, analyses can be constructed that explicitly characterize the behaviors on which their results are unsound, e.g., [13]. When an analysis does this, it provides a more accurate characterization of the evidence it contributes regarding property-behavior conformance.

The second scenario we consider involves *generalizing partial evidence* produced by analyses that focus just on subsets of program behavior. For instance, when a test suite is executed (the series of dots at the bottom of Figure 2) the individual tests check for the property via an oracle. While the input values of each test force the execution of a system behavior it is well understood that other input values may force exactly the same behavior, i.e., the same sequence of branches. Recent work on dynamic symbolic execution [16] illustrates one method for *generalizing partial evidence* produced by a dynamic analysis, such as testing, to produce a broader characterization of the behavioral coverage of the analysis. Another example of such generalization is work on multi-threaded testing that infers coverage of all executions sharing the same logical schedule from a single observed execution [11].

The third scenario we consider focuses on properties rather than behaviors. When it is difficult to reason about a given property, due to lack of cost-effective V&V techniques, one can aim to pro-

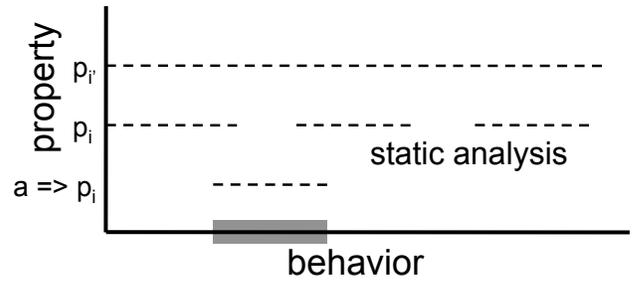


Figure 3: The lines depict the behavior coverage provided by a static analyses for variants of program property p_i

duce partial evidence by deliberately *weakening* the property to be checked. Figure 3 illustrates two ways this might be achieved.

The middle line illustrates the partial evidence produced when analyzing a system for property p_i . If p_i requires the analysis of complex properties of system data, e.g., [15], then the analysis is unlikely to succeed completely; for such properties the cost of applying test oracles may even become significant. When presented with such a property a common approach in the static analysis literature is to focus on related properties that are more control-oriented, e.g., [2, 7].

We illustrate two forms of property weakening relative to p_i . The top line in the Figure illustrates how a variant property p_i' might yield broader behavior coverage. Ideally one would be able to relate these properties such that $\exists p_i'' : p_i' \otimes p_i'' \implies p_i$ for some operator \otimes . A second form of property weakening for analysis and testing comes indirectly through the incorporation of assumptions about system behavior. The bottom line in the Figure illustrates the analysis of p_i when program behavior is restricted to conform to assumption a (the shaded region of the x-axis).

A number of researchers have explored approaches to scaling advanced V&V techniques to large systems with complex correctness properties. Some notable successes in this regard have resulted from a willingness to compromise the ideal application of a technique – by introducing unsoundness or imprecision into the analysis or by weakening the property under analysis. Rather than view compromise as a negative, we regard it as a path towards progress as long as the partial evidence of system property-behavior correspondence can be captured accurately and comprehensively.

4. CHALLENGES AND OPPORTUNITIES

We believe that the software V&V community is well-positioned to look beyond the boundaries of individual analysis, verification, and testing techniques to consider the larger question of how the individual techniques fit together to provide an explicit body of evidence about software system quality.

In our experience, we have seen numerous instances where even a “failed” application of a V&V technique produced some evidence of property-behavior conformance. By accumulating evidence from multiple techniques, a comprehensive assessment of property-behavior coverage can be pieced together. If that combined evidence can be presented to developers and users of systems they will be better equipped to make decisions about whether to deploy the system, how to deploy the system, and what additional V&V ought to be applied.

We conclude with a series of questions, with associated comments, that illustrate some of the challenges and opportunities that lie along this line of inquiry.

What are the requirements for V&V evidence? We believe that there should be a standardized syntax and semantics for evidence to enable tools to interchange and process evidence. Meta-data, such as a record of the provenance of the evidence, i.e., the tools that contributed to it and processed it, will also be necessary to enable the reproduction or auditing of the evidence.

What frameworks for encoding property-behavior coverage are appropriate for capturing evidence from a variety of V&V techniques? While there are a number of possible candidates, we believe that logical representations offer some compelling advantages that are worth considering. For example, a wide-range of automated verification techniques operate by generating verification conditions as logical formula, symbolic execution techniques construct logical representations of path conditions, and specification-based approaches may start with pre/post conditions expressed using logic. Moreover, the use of logical path conditions provides a bridge between traditional white-box testing and specification-based techniques.

How can existing analysis and testing techniques be adapted to report partial property-behavior coverage? V&V techniques are “stressed” as it is to scale to the large complex software systems being developed today. Adding the requirement that they produce evidence would only seem to further reduce their applicability in practice. One approach to consider is to develop techniques that operate in different modes. For instance, an analysis running in “fault finding mode” might be explicitly unsound for scalability and be applied earlier in development. Once it fails to find any more faults the analysis is reconfigured to “proof mode” in which it maximizes soundness to prove property-behavior conformance. Finally, it is run a single time in “evidence generating mode” to extract as much partial property-behavior coverage as possible.

When using partial V&V evidence what criteria should be used to design the property space? It seems important to consider the effectiveness of different V&V techniques on types of properties. In fact, a given property might have evidence produced from a set of techniques, so taking into account the classes of behaviors relevant to a property and amenable to a particular technique might also be warranted. Often times one views redundancy in a specification as undesirable, but with partial evidence one might find that V&V of a set of (partially) redundant properties, which are related to a given target property, provides an opportunity for greater coverage of the target property. Given such considerations, what constitutes a sufficient set of properties.

How can property-behavior coverage be combined and put to effective use? Combining evidence produced by multiple V&V techniques offers the opportunity for more effective and more useful combination of different techniques. In Figure 2, the gray shaded areas on the x-axis depict the regions of behavior that are not covered by either the partial evidence produced by the analysis or the generalized evidence produced by the test suite. That information can be used to target additional V&V activities. For example, assumption a in Figure 3 might be calculated from the gap in coverage for property p_i at some point during the V&V process.

The notion of combined evidence offers an additional potential benefit. The strengths of one V&V technique can be applied to “mask” the weakness or unsoundness of another. For example, a static analysis might be applied to confirm a property on all behaviors that do not involve calls to foreign functions. The behaviors that do involve such calls might then be targeted by testing techniques which have no problem executing foreign function calls.

How can property-behavior coverage be presented for human consumption and for consumption by other V&V to target their application? The property-behavior relation is huge and representing it in a form that can be easily understood by a developer, regulator, or user is a daunting challenge. We believe that it will be necessary to develop means of not only combining evidence but also of generalizing it wherever possible to simplify its uptake by system stakeholders. Generalization, perhaps of a different form, may also prove valuable in making the processing of evidence by other tools more tractable.

5. REFERENCES

- [1] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Conf. on Obj. Oriented Prog. Sys. Lang. and App.*, pages 589–608, 2007.
- [2] T. Ball and S. K. Rajamani. The SLAM Toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer, 2001.
- [3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.
- [4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 196–207, 2003.
- [5] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI*, pages 209–224, 2008.
- [6] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *Conf. on Obj. Oriented Prog. Sys. Lang. and App.*, pages 569–588, 2007.
- [7] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Int’l. Conf. on Soft. Eng.*, pages 411–420, May 1999.
- [8] M. Dwyer and R. Purandare. Residual dynamic typestate analysis. In *Int’l. Conf. on Aut. Soft. Eng.*, pages 124–133, 2007.
- [9] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *Int’l. Symp. Softw. Test. Anal.*, pages 133–144, 2006.
- [10] D. Jackson, M. Thomas, and L. I. Millett. Software for dependable systems : Sufficient evidence? Technical report, National Research Council of the National Academies, 2007.
- [11] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Computers*, 36(4):471–482, 1987.
- [12] J. Oliver. *The Incomplete Guide to the Art of Discovery*. Columbia University Press, 1991.
- [13] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the SIGSOFT Symposium on Foundations of Software Engineering*, 2008.
- [14] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proc. 30th Intl. Conf. on Soft. Eng.*, pages 341–350, 2008.
- [15] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
- [16] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proc. 10th Euro. Soft. Eng. Conf. held jointly with 13th ACM SIGSOFT Intl. Symp. on Found. of Soft. Eng.*, pages 263–272, 2005.
- [17] S. Srivastava, S. Gulwani, and J. S. Foster. Vs3: Smt solvers for program verification. In *Proceedings of Computer Aided Verification*, 2009.
- [18] R. Vallée-Rai. SOOT: A Java bytecode optimization framework. Master’s thesis, School of Computer Science, McGill University, Montreal, Canada., Oct 2000.
- [19] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE*, Sept. 2000.
- [20] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.