# We Need a Testability Transformation Semantics

Mark Harman[1,2(✉)]

[1] Facebook, London, UK
mark.harman@ucl.ac.uk
[2] University College, London, UK

**Abstract.** This paper (This paper is a brief outline of some of the content of the keynote by the author at the $16^{th}$ International Conference on Software Engineering and Formal Methods (SEFM 2018) in Toulouse, France; 27th–29th June 2018.) briefly reviews Testability Transformation, its formal definition, and the open problem of constructing a set of formal test adequacy semantics to underpin the current practice of deploying transformations to help testing and verification activities.

## 1 Introduction

Testability transformation modifies a program to make it easier to test. Unlike traditional program transformation [18,43], which alters a program's syntax without changing its input–output behaviour, Testability Transformation may alter functionality. Nevertheless, it does respect a program semantics, defined by the test adequacy criterion. Therefore, the sense in which a Testability Transformation is meaning preserving rests on a formal definition of the semantics of transformations that preserve test adequacy. Sadly, to date, such a test adequacy semantics has yet to be formally defined.

There are several widely-used test adequacy criteria in practice, such as statement, branch, data flow, path and mutation adequacy [8,9,31]. Each of these adequacy criteria gives rise to a different test adequacy semantics, the definitions of which and their relationship as a formal lattice of semantics remain interesting and important open scientific problems at the intersection of Software Engineering and Formal Methods (SEFM). Tackling this set of related semantic definitions will provide a firm mathematical foundation for Testability Transformation, and by extension, may also yield insights into currently-deployed testing techniques. This paper aims to draw out some of these open problems as research questions for the SEFM research community.

Testability Transformation itself, was first formalised in 2004 [25]. However, informally, software engineers have performed transformations to aid testability for considerably longer. For example, it is routine for testers to 'mock up' procedures to allow testing of the 'whole' before the 'parts' have been fully implemented. Similarly, for verification purposes, it is often necessary to model parts of the system with stubs, where source code is unavailable for analysis [2].


E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 3–17, 2018.
https://doi.org/10.1007/978-3-319-92970-5_1

These simple mock–and–model transformations are relatively well-understood. By contrast, other transformations, that have the potential to more dramatically benefit the effectiveness and efficiency of testing and verification, remain less well-understood, because of a lack of proper formal underpinnings. These more ambitious Testability Transformations exploit the way in which transformation rules can be 'more aggressive' in their disruption of the input–output behavior, so long as they take care to preserve test adequacy semantics.

For example, Testability Transformation has been used to tackle problems such as flag variables [4,5,20], unstructured control flow [28], data flow testing [34], state-based testing [33], and nesting [40]. An overview of these applications of Testability Transformation can be found in the 2008 survey [23]. As well as source code transformation, it has also been applied, more recently at bytecode level [37]. It has also been applied to alleviate problems in dynamic symbolic execution [11] and to tackle the oracle problem in testing [39].

Nevertheless, the lack of formal underpinning means that much of this work rests on, as yet, uncertain foundations and this inhibits progress; transformations should take care to preserve test adequacy semantics, *yet there is no formal description* of test adequacy semantics for any of the widely-used adequacy criteria. Much more could undoubtedly be achieved in terms of practical transformation benefit, if only the designers of Testability Transformations had a formal test adequacy semantics to which they could appeal. Practitioners and researchers could use such a semantics to motivate, justify or otherwise explore their transformation spaces.

## 2 Formal Definition of Testability Transformation

The goal of a Testability Transformation is to make it easier to test the untransformed program. Although test data is ultimately applied to the original program, the technique that generates it uses the transformed version because it is easier. The transformation process therefore needs to guarantee that any adequate set of test data, generated and adequate for the transformed program, will also be adequate for the original program. This (test-based) meaning preservation guarantee replaces the more familiar guarantee, whereby traditional transformation preserves the input–output relationship of the untransformed program.

The formal definition of Testability Transformation is simple and has been known for some time [25]. In this section, we briefly review this formal definition of Testability Transformation, based on earlier work [22,25], which we augment by defining, slightly more formally, what is intended by the term 'test adequacy criterion':

**Definition 1 (Test Adequacy Criterion).** Let $\mathbf{P}$ be a set of programs, with an input space $\mathcal{I}$. A test adequacy criterion, $c$ is a function from programs to predicates over sets of inputs:

$$c \in P \rightarrow 2^{\mathcal{I}} \rightarrow \{true, false\}$$

A test adequacy criterion determines whether a set of test inputs (i.e., test suite) meets the criterion of interest for the tester. An example is the branch adequacy criterion which is true, for test suite $T$ and program $P$, when for every feasible branch, $b$ in $P$, the test suite $T$ contains at least one test that executes $b$. Another example is the statement adequacy criterion, which is true when every reachable statement is executed by the at least one test in the test suite.

In practice [3], since branch traversibility and statement reachability are both undecidable, testers settle for a measure of the percentage of branches (statements) covered, setting (rather arbitrary) percentage thresholds for satisfaction of the coverage predicate. Alternatively, practicing testers may choose to identify a set of important branches (statements) that are know to be feasible (and of interest) that would need to be covered in order for the test adequacy criterion to return true.

With the concept of tests adequacy criterion in hand, we can now define what it means to be a transformation that is concerned with both programs *and* the tester's chosen adequacy criterion, both of which might be transformed, as we shall see.

**Definition 2 (Testing-Oriented Transformation).** Let $\mathbf{P}$ be a set of programs and $\mathbf{C}$ be a set of test adequacy criteria. A program transformation is a partial function in $\mathbf{P} \rightarrow \mathbf{P}$. A *Testing-Oriented Transformation* is a partial function in $(\mathbf{P} \times \mathbf{C}) \rightarrow (\mathbf{P} \times \mathbf{C})$.

The test adequacy criterion, $\mathbf{C}$ refers to the overall criterion, which may be composed of a set of instances, each of which is denoted by lower case $c$. For instance, branch coverage is a possible choice for $\mathbf{C}$, while a particular instance, $c$, might capture the specific set of branches to be covered.

A testing-oriented transformation is defined to be a *partial* function, simply to allow that the transformation might be undefined for some programs; the possibility that a testing-oriented transformation may fail to terminate is not ruled out. In practice, this is relatively unimportant generalisation from total functions, because it will always be acceptable to leave the program untransformed. Therefore, a partial testing-oriented transformation can be easily converted to a total testing-oriented transformation.

**Definition 3 (Testability Transformation).** A Testing-Oriented Transformation, $\tau$ is a *Testability Transformation* iff, for all programs $p$, criteria $c$, and test sets $T$, $T$ is adequate for $p$ according to $c$ if $T$ is adequate for $p'$ according to $c'$, where $\tau(p, c) = (p', c')$.

In some cases, the test adequacy criterion can be transformed along with the program under test and this definition allows for that. We call this a co-transformation (that is, one in which both the program under test and the test adequacy criteria are transformed). For instance, MC/DC test adequacy [49] can be safely co-transformed to branch adequacy provided that the program is co-transformed to expand the boolean logic operator terms in predicates [8]. This co-transformation means that branch coverage of the expanded program

is equivalent (has the same test adequate test input sets) as MC/DC on the untransformed version.

In other circumstances, it will be more convenient to leave the adequacy criterion untransformed, and thereby to produce a program transformation that respects it. That is, for some criterion, $c$, a $c$–preserving Testability Transformation guarantees that the transformed program is suitable for testing with respect to the original criterion; a widely-used scenario in practice.

**Definition 4 ($c$–Preserving Testability Transformation).** Let $\tau$ be a Testability Transformation. If, for some criterion, $c$, for all programs $p$, there exists some program $p'$ such that $\tau(p, c) = (p', c)$, then $\tau$ is called a $c$–preserving Testability Transformation.

As an illustrative example, consider the program

**Example 1.** *Simple program under test illustration*

```
input(z);
x=1; y=z;
if (y>3) x=x+1;
    else x=x-1;
output(x,y,z)
```

Example 1 can be transformed to

```
input(z)
if (z>3) ;
    else ;
output(x,y,z)
```

Such a transformation does not preserve the effect of the original program on the variables `x` and `y` and, therefore, does not preserve the input–output behaviour of Example 1. However, it does preserve the set of sets of inputs that execute (or 'cover' in testing nomenclature) all branches. It also preserves the set of sets of inputs that cover all statements, despite the fact that the transformation process has removed some statements. Therefore, the transformation can be said to be a 'branch–adequate transformation'. It is also a 'statement–adequate transformation'.

This guarantee of test-adequacy for the transformation has practical significance: test inputs can be constructed from the transformed program, safe in the knowledge that any set of inputs that is branch adequate for the transformed program will also be branch adequate for the original (Example 1). Since branch adequacy subsumes statement adequacy, we would also immediately know that such a set of test inputs would also be statement adequate for Example 1.

Of course, when executed on each version, the input–output relationship will be different, but this is unimportant for test input generation; the test inputs, once generated, will be applied to Example 1, *not* the transformed version.

The advantage of the transformation is that it may prove to be more effective and/or efficient to generate tests from the transformed program than the original. This has been the motivation for the previous work on Testability Transformation, which has revealed many such cases whereby transformation eases test input generation and verification.

Testability transformation is not only useful, but also offers interesting intellectual and scientific challenges: transformations can cover different paths in the transformed program, yet still preserve path adequacy, because the sets of test inputs that cover all paths in the transformed program also cover all the paths in the original. The behaviour of transformed programs and their relationship to the original from which they are constructed are thus highly subtle and a full understanding clearly necessitates a proper formal semantic treatment. As a simple illustration of this need, consider the simple fragment $F$, defined below:

**Example 2. (F)**

```
input(x);
if (x>0) x=-1;
    else x=1;
output(x)
```

The fragment $F$ can be branch-adequately transformed to $F'$ defined below:

**Example 3. (F')**

```
input(x);
if (x>0);
    else;
output(x)
```

This transformation is also path-adequate. Furthermore, $F \circ F$ can be path-adequately transformed to $F' \circ F'$. That is, as illustrated in Fig. 1, test suites that cover all branches of $F' \circ F'$, also cover all branches of $F \circ F$, even though the test inputs will traverse different branches. Indeed, test inputs that cover all paths in $F' \circ F'$ *also* cover all paths in $F \circ F$, even though they follow *different* paths in the transformed program than they follow in the original.

Formal treatments of slicing may provide a starting point for understanding this behaviour [7,13,42,46], since we are partially concerned with the interplay between data and control flow. Slicing makes a good starting point for understanding a subclass of Testability Transformations for two reasons:

1. Slicing reduces program size and is, thereby, likely to reduce execution time, which may, in turn, help improve the performance of test techniques [24]. Slicing will likely be particularly useful for test techniques that require repeated execution of the program under test to generate test inputs [12,26], because the slice computation cost can be offset by the multiple test execution time gain.

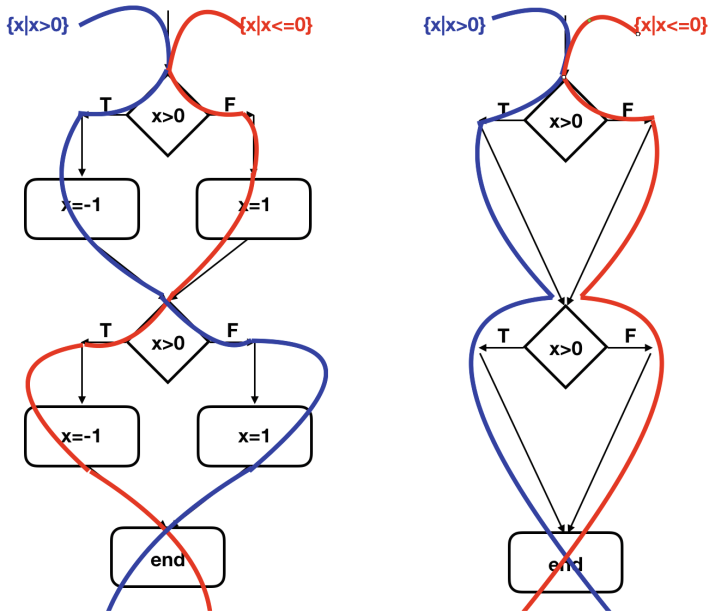


**Fig. 1.** The version on the right is both a branch-adequate and a path-adequate transformation of the version on the left, even though all possible test inputs follow different paths in each program. The original program partitions the set of all inputs (all initial values of `x`) into two disjoint subsets, each of which causes execution to follow one of two paths. Two test inputs, one from each subset, are necessary and sufficient to cover all (feasible) paths. The transformed program creates the same test input partition and so it preserves path-adequacy. This is illustrated by the input sets located at the top of each figure, and the path that inputs drawn from each follows. These two paths for each program are depicted in red and blue (or light and dark grey when printed in black-and-white). Observe that the feasible path sets are different for the two programs, and these two disjoint input subsets therefore follow different paths in each program. However, all input sets that cover (follow) all feasible paths in the transformed program on the right, nevertheless, also follow all feasible paths in the original on the left; the transformation is path adequacy preserving.

2. Slicing on all predicates for which we seek branch adequacy will maintain the computation needed for these predicates to correctly replicate their behaviour in the unsliced original program, thereby ensuring sufficiency of branch adequacy for these predicates.

However, although it may sufficient for path-adequacy, preserving control flow does not appear to be *necessary*, as illustrated in Fig. 1.

## 3 Testability Transformation and Abstract Interpretation

There exist valid Testability Transformations for which the allowable transformations are neither subsets nor supersets the traditional transformation set

that preserves the input–output behaviour of the untransformed program. For instance, consider this simply illustrative example:

**Example 4 (More Concrete).** `if (x>y) ; else ;`

This program can be transformed to the empty program, while preserving the input–output behaviour of untransformed program. However, this apparently 'simple optimization' does not preserve the sets of test suites that cover all branches of the original and, therefore, it cannot be a branch–adequate transformation. Therefore, there exist Testability Transformation semantics that are clearly not simply more abstract (allow a superset of transformations) than the conventional input–output relation semantics.

Furthermore, consider the program:

**Example 5 (More Abstract).** `if (x>y) x=1; else x=2;`

This program can be branch–adequately transformed to:

```
if (x>y) ; else ;
```

Clearly, such a branch–adequate transformation does not preserve the conventional input–output semantics. Therefore, branch–adequacy semantics is also not more concrete than conventional semantics.

From Examples 4 and 5 we are forced to conclude that branch–adequacy semantics is neither universally more concrete nor more abstract than conventional input–output relation semantics. Similar arguments can be made for other test adequacy criteria and the semantics they denote.

However, this observation does not mean that the consideration of the abstraction level of the semantics preserved has no role to play in formalising Testability Transformation. Quite the opposite: it should be possible to construct a lattice of Testability Transformation semantics ordered by semantic abstraction [22]. Therefore, abstract interpretation [16] would also be a promising framework within which to explore test adequacy semantics.

Perhaps a trace-based semantics would be more suitable to capture the properties preserved by Testability Transformation. However, as the example in Fig. 1 demonstrates, the traces followed in two different programs can be entirely different and yet the two programs are, nevertheless, path adequate testability transformations of each other. This suggests that perhaps merely abstracting from simple traces may prove insufficient to capture the semantics of test adequacy[1].

[1] Author's note: I sincerely hope that I may be proved wrong in this conjecture, since a simple abstracted trace semantics that captures test adequacy (and thereby informs Testability Transformation) would shed much light on many testing problems.

## 4 Testability Transformation and Metamorphic Testing

Metamorphic testing [15] is one approach to tackle the test oracle problem [6]. The oracle problem is captured by the question:

"What output(s) should be expected for a given input?"

With metamorphic testing, the tester uses relationships between already observed test cases (input–output pairs) and properties of a as-yet-unseen outputs for some newly provided test input. In this way Metamorphic Testing offers an oracle, albeit an incomplete one, where either no oracle, or an even more incomplete oracle, was previously available. As a simple illustration we might have:

$$\text{if } p(i) = r \text{ then } p(f(i)) = g(r)$$

for some program $p$ and so-called 'metamorphic relations' $f$ and $g$.

In this way we define one test case $(f(i), g(r))$ in terms of another, $(i, r)$, using metamorphic relations ($f$ and $g$).

As a more concrete example, suppose our first test reveals that

$$\texttt{CheckBalance}(i) = r,$$

giving us output $r$, a balance in a bank account, for the input $i$, a bank account number. From this test case, we can generate a set of new tests, using a non-negative deposit, $x$:

$$\texttt{CheckBalance}(\texttt{Deposit}(i, x)) = r', \text{where } x \geq 0,$$

such that we expect the condition $r' \geq r$. That is, we do not know the correct value $r'$ should take (because we lack a complete test oracle). Nevertheless, we do have a *property* that $r'$ should satisfy, for this new input $x$, expressed in terms of some previously witnessed test case $(i, r)$.

A far more detailed and complete account of metamorphic testing can be found elsewhere [15]. For the present paper we merely wish to observe that metamorphic testing already lies at an intersection between Software Engineering and Formal Methods, because there is clearly a need to formalise the metamorphic relations for each metamorphic application area in order to provide a sound foundation for each.

It is also interesting to observe the close resemblance between metamorphic test relations and algebraic data type specifications [21]; both essentially capture a set of algebraic properties to be maintained in all valid instances. For instance, in the `CheckBalance` example above, metamorphic testing is simply exploiting the following algebraic property of `CheckBalance` and `Deposit` to find new test cases for old:

$$\begin{gathered} \forall x \in \mathbb{R}. \\ x \geq 0 \ \wedge \ \texttt{CheckBalance}(i) = r \ \wedge \ \texttt{CheckBalance}(\texttt{Deposit}(i, x)) = r' \\ \Rightarrow \\ r' \geq r \end{gathered}$$

which has a similar structure (and potential uses) as the familiar algebraic properties of data structures, such as:

$$\forall s \in \texttt{Stack}, x \in \texttt{Elem}.$$
$$\texttt{top}(\texttt{push}(x, s)) = x$$

There are at least two connections between Testability Transformation and Metamorphic Testing:

1. **Metamorphic Testing as Testability Transformation:** Consider again the metamorphic relations $f$ and $g$ for the program $p$, where

$$\text{if } p(i) = r \text{ then } p(f(i)) = g(r)$$

   If we re-write this, slightly, as

$$\text{if } p(i) = r \text{ then } (f \circ p)(i) = g(r)$$

   it becomes immediately clear that $f \circ p$ can be regarded as a (relatively simply) transformed version of $p$, and that the transformation of the oracle from $p(i) = r$ to $(f \circ p)(i) = g(r)$ can also be regarded as a transformation of the oracle, thereby resembling a test adequacy criterion transformation.
2. **Metamorphic Testability Transformation:** If we generalise a metamorphic equation like if $p(i) = r$ then $p(f(i)) = g(r)$, we can obtain a kind of 'Metamorphic Testability Transformation', which combines the metamorphic effect of the oracle with the transformation effect on the program. The generalisation would be if $p(i) = r$ then $(\mathcal{T}_{\text{p}}(p))(f(i)) = (\mathcal{T}_{\text{c}}(g))(r)$ where $\mathcal{T}_{\text{p}}$ is a Testability Transformation on programs and $\mathcal{T}_{\text{c}}$ is a Testability Transformation on the oracle; akin to a testability co-transformation of program and test adequacy criterion, as defined by Definition 3 of Testability Transformation.

These two possibilities for combining Metamorphic Testing and Testability Transformation are each, essentially, formalisations of McMinn's Co-testability Transformation, which transforms both the program under test and the corresponding oracle for that program under test [39].

## 5 Testability Transformation Research Questions to Be Tackled Using Formal Methods

In this section we give seven open problems concerning formal aspects of Testability Transformation, each of which could yield sufficient challenges for a connected set of research activities involving several non-trivial subprojects, perhaps suitable for a PhD programme or other grant-funded project at the intersection of Software Engineering and Formal Methods.

**Research Question 1 (Novel Semantics).** Can we define an elegant formal semantics for Branch Adequacy?

According to such a branch adequacy semantics, any meaning–preserving transformation would preserve the sets of sets of branch adequate inputs. Ideally, the semantics should admit specialisation to a specific set of branches, so there is also a need to formalise the specification of branches to support such a specialisation.

Branch adequacy is widely-studied and held out as a goal for many test techniques [10,45], making this good starting point. Progress on Research Question 1 alone could yield great progress in our understanding of many test input generation techniques.

There are several possible candidates for defining a test adequacy semantics. Different notations may be more valuable in different contexts. For instance, one might presume that algebraic semantics [19] would be intellectually close to the notions of transformation and might be best suited to declarative formulation and correctness proofs of transformation rules. By contrast, operational semantics [44] might best explain and inform the way in which test adequacy criteria have an operational character, being concerned with execution and, for several practically important criteria, execution paths. Finally, Hoare triples [1,29] might best capture the way that certain aspects of state must be maintained in order to preserve test behaviour, while others can be safely removed without affecting test adequacy.

**Research Question 2 (Semantic Abstraction).** What is the relationship between Testability Transformation and Abstract Interpretation?

As discussed in Sect. 3, Abstract Interpretation appears to be clearly relevant to Testability Transformation, but not to explain the relationship between test adequacy semantics and conventional semantics. Rather, it would appear that each of the different test adequacy criteria gives rise to a different semantics. It is the relationship between these different test adequacy semantics that may best be explained in terms of Abstract Interpretation.

An initial answer to Research Question 2 might start by considering the relationships between the test adequacy semantics for branch, statement and path coverage, all of which exhibit a *prima facie* structural relationship. It will also be interesting to compare the abstraction relationships between test adequacy criteria and the subsumption relationship between them; perhaps abstract interpretation can also provide a semantic framework for better understanding of test adequacy subsumption.

**Research Question 3 (Mutation Semantics).** What is the test adequacy semantics for mutation testing?

Mutation testing [31] has been shown to be highly effective as a test adequacy criterion [14,32]. Therefore, it would be interesting to formalise the semantics preserved by mutation testing. There are different forms of mutation, such as strong, firm and weak [48]; a formalisation would allow a more rigorous investigation of relationships between these different forms of mutation. It would also be useful to specialise a semantics, so that it can apply only to a specific set

of mutants, thereby facilitating a formal investigation of relationships between mutations sets and mutant subsumption [30,35].

Finally, since mutants are, themselves, transformations of the program under test, it would be fascinating to formalise the mutants themselves, and candidate fixes, within the same Testability Transformation formal semantic framework. This would facilitate a mathematical investigation of mutants as repair candidates [36], mutant equivalence [27,38,41], links between mutation and metamorphic testing [50], and of the empirically-observed phenomenon of mutational robustness [47].

**Research Question 4 (Transformation Sets).** What are the transformation rules that are correct for a given test adequacy criterion (branch, mutation...) and what proof obligations can be discharged when defining them?

Once some initial theoretical foundations have been laid, it would be highly useful for testing practitioners to have such sets of transformation rules. Formal semantic foundations imbue transformation sets with the safety-in-application that comes for formal guarantees of meaning preservation (with respect to a test adequacy criterion of interest to the tester). Work on abstract interpretation frameworks for specifying and reasoning about the correctness of transformation rules [17] may be useful here.

**Research Question 5.** What transformations can be performed on test adequacy criteria and what proof obligations do they raise?

Test adequacy criteria can be transformed, both with and without transforming the program under test. This has been known for some time (the example of MC/DC to branch coverage from Sect. 2 is one such example). However, we lack a formal framework within which to understand such criteria transformations with which we could attempt to prove their correctness.

**Research Question 6.** Can we define a formal semantics of Testability Transformation that helps us to better inform and understand the practice of Metamorphic Testing?

A formal understanding of this relationship would help to generalise both Testability Transformation and Metamorphic Testing, as set out in Sect. 4. Such a formalisation would also provide a new mathematical foundation for the study of metamorphic testing, using the formal semantics established in answer to Research Question 1. This would not only be theoretically helpful (yielding improved understanding and reasoning about Metamorphic Testing), it would also have practice benefit by extending the reach of Metamorphic Testing, to include transformed programs as well as transformed test oracles.

**Research Question 7 (Anticipatory Testing).** What Testability Transformations can be used to assist and pre-harden programs for Anticipatory Testing?

Anticipatory Testing is a (very recent) new approach to testing proposed by Tonella[2] that seeks to identify tests that reveal future failures in systems before they have occurred. A transformation that makes it easier to find anticipatory tests would help the tester to find such cases and may alleviate some of the challenges of anticipatory testing.

It will also be interesting to explore Anti-Testability Transformations; transformations that make the task of finding anticipatory tests *harder*. Using a more conventional meaning-preserving transformation approach (in which the transformed program is considered to be a meaning–preserving, and improved, version of the original), such anticipatory-test-denying transformations could yield a form of automated system security hardening.

## 6 Conclusion

This paper's primary role is to argue for a formal semantic underpinning for Testability Transformation. It outlines how such a semantic framework could combine practical industrial impact with theoretical elegance and intellectual challenge. The hope is that the research community will take up the challenge of defining these much-needed formal semantics.

The definition of the sets of semantics preserved by different test adequacy criteria will undoubtedly also shed much needed light on test adequacy more generally, and thereby provide a mathematical basis for much of the current *ad hoc* practices of software testing.

**Acknowledgements.** Many thanks to Patrick Cousot, Paul Marinescu, Peter O'Hearn, Tony Hoare, Mike Papadakis, Shin Yoo, and Jie Zhang for comments on earlier drafts. Thanks also to the Facebook Developer Infrastructure leadership for their support and to the European Research Council for part-funding my scientific work through the ERC Advanced Fellowship scheme.

## References

1. Apt, K.R.: Ten years of Hoare's logic: a survey - part I. ACM Trans. Prog. Lang. Syst. **3**(4), 431–483 (1981)
2. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Proceedings of the First European Systems Conference (EuroSys 2006), pp. 73–85. Leuven, Belgium, April 2006
3. Bardin, S., Delahaye, M., David, R., Kosmatov, N., Papadakis, M., Traon, Y.L., Marion, J.Y.: Sound and quasi-complete detection of infeasible test requirements. In: International Conference on Software Testing, Verification and Validation (ICST 2015), pp. 1–10. IEEE Computer Society (2015)

[2] https://www.pre-crime.eu.

4. Baresel, A., Binkley, D., Harman, M., Korel, B.: Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In: International Symposium on Software Testing and Analysis (ISSTA 2004), pp. 108–118. Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes **29**(4)
5. Baresel, A., Sthamer, H.: Evolutionary testing of flag conditions. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L.D., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A.C., Dowsland, K.A., Jonoska, N., Miller, J. (eds.) GECCO 2003, Part II. LNCS, vol. 2724, pp. 2442–2454. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45110-2_148
6. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The Oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015)
7. Barraclough, R., Binkley, D., Danicic, S., Harman, M., Hierons, R., Kiss, A., Laurence, M.: A trajectory-based strict semantics for program slicing. Theor. Comput. Sci. **411**(11–13), 1372–1386 (2010)
8. Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold, New York (1990)
9. Bertolino, A.: Software testing research: achievements, challenges, dreams. In: Briand, L., Wolf, A. (eds.) Future of Software Engineering 2007. IEEE Computer Society Press, Los Alamitos (2007)
10. British Standards Institute: BS 7925–2 software component testing (1998)
11. Cadar, C.: Targeted program transformations for symbolic execution. In: 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 906–909 (2015)
12. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
13. Cartwright, R., Felleisen, M.: The semantics of program dependence. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 13–27 (1989)
14. Chekam, T.T., Papadakis, M., Traon, Y.L., Harman, M.: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, 20–28 May 2017, pp. 597–608 (2017)
15. Chen, T.Y., Feng, J., Tse, T.H.: Metamorphic testing of programs on partial differential equations: a case study. In: 26th Annual International Computer Software and Applications Conference (COMPSAC 2002), pp. 327–333. IEEE Computer Society (2002)
16. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Logic Comput. **2**(4), 511–547 (1992)
17. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: The 29th ACM Symposium on Principles of Programming Languages (POPL 2002), pp. 178–190, Portland, Oregon, 16–18 January 2002
18. Darlington, J., Burstall, R.M.: A transformation system for developing recursive programs. J. Assoc. Comput. Mach. **24**(1), 44–67 (1977)
19. Goguen, J.A., Malcolm, G.: Algebraic Semantics of Imperative Programs. MIT Press, Cambridge (1996)
20. Gong, D., Yao, X.: Testability transformation based on equivalence of target statements. Neural Comput. Appl. **21**(8), 1871–1882 (2012)
21. Guttag, J.: Abstract data types and the development of data structures. Commun. ACM **20**(6), 396–404 (1977)

22. Harman, M.: Open problems in testability transformation (keynote paper). In: 1st International Workshop on Search Based Testing (SBST 2008), Lillehammer, Norway (2008)
23. Harman, M., Baresel, A., Binkley, D., Hierons, R., Hu, L., Korel, B., McMinn, P., Roper, M.: Testability transformation – program transformation to improve testability. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 320–344. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_11
24. Harman, M., Danicic, S.: Using program slicing to simplify testing. Softw. Test. Verification Reliab. **5**(3), 143–162 (1995)
25. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Softw. Eng. **30**(1), 3–16 (2004)
26. Harman, M., Jia, Y., Zhang, Y.: Achievements, open problems and challenges for search based software testing (keynote paper). In: 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015), Graz, Austria, April 2015
27. Harman, M., Yao, X., Jia, Y.: A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: 36th International Conference on Software Engineering (ICSE 2014), pp. 919–930, Hyderabad, India, June 2014
28. Hierons, R., Harman, M., Fox, C.: Branch-coverage testability transformation for unstructured programs. Comput. J. **48**(4), 421–436 (2005)
29. Hoare, C.A.R.: An axiomatic basis of computer programming. Commun. ACM **12**, 576–580 (1969)
30. Jia, Y., Harman, M.: Higher order mutation testing. J. Inf. Softw. Technol. **51**(10), 1379–1393 (2009)
31. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
32. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: International Symposium on Foundations of Software Engineering (FSE), pp. 654–665 (2014)
33. Kalaji, A., Hierons, R.M., Swift, S.: A testability transformation approach for state-based programs. In: 1st International Symposium on Search Based Software Engineering (SSBSE 2009), pp. 85–88. IEEE, Windsor, May 2009
34. Korel, B., Harman, M., Chung, S., Apirukvorapinit, P., Gupta, R.: Data dependence based testability transformation in automated test generation. In: 16th International Symposium on Software Reliability Engineering (ISSRE 2005), pp. 245–254, Chicago, Illinios, USA, November 2005
35. Kurtz, B., Ammann, P., Delamaro, M.E., Offutt, J., Deng, L.: Mutant subsumption graphs. In: 10th Mutation Testing Workshop (Mutation 2014), Cleveland Ohio, USA, March 2014, to appear
36. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
37. Li, Y., Fraser, G.: Bytecode testability transformation. In: Cohen, M.B., Ó Cinnéide, M. (eds.) SSBSE 2011. LNCS, vol. 6956, pp. 237–251. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23716-4_21
38. Madeyski, L., Orzeszyna, W., Torkar, R., Jozala, M.: Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation. IEEE Trans. Softw. Eng. **40**(1), 23–42 (2014)
39. McMinn, P.: Search-based failure discovery using testability transformations to generate pseudo-oracles. In: Rothlauf, F. (ed.) Genetic and Evolutionary Computation Conference (GECCO 2009), pp. 1689–1696. ACM, Montreal (2009)

40. McMinn, P., Binkley, D., Harman, M.: Empirical evaluation of a nesting testability transformation for evolutionary testing. ACM Trans. Softw. Eng. Methodol. **18**(3) (2009). Article no. 11
41. Papadakis, M., Jia, Y., Harman, M., Traon, Y.L.: Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: 37th International Conference on Software Engineering (ICSE 2015), pp. 936–946, Florence, Italy (2015)
42. Parsons-Selke, R.: A graph semantics for program dependence graphs. In: Sixteenth ACM Symposium on Principles of Programming Languages (POPL), Austin, TX, 11–13 January 1989, pp. 12–24 (1989)
43. Partsch, H.A.: The Specification and Transformation of Programs: A Formal Approach to Software Development. Springer, Heidelberg (1990)
44. Plotkin, G.D.: The origins of structural operational semantics. J. Logic Algebraic Prog. **60**, 3–15 (2004)
45. Radio Technical Commission for Aeronautics: RTCA DO178-B Software considerations in airborne systems and equipment certification (1992)
46. Reps, T., Yang, W.: The semantics of program slicing. Technical report 777, University of Wisconsin (1988)
47. Schulte, E., Fry, Z.P., Fast, E., Weimer, W., Forrest, S.: Software mutational robustness. Genet. Program. Evolvable Mach. **15**(3), 281–312 (2014)
48. Woodward, M.R., Halewood, K.: From weak to strong, dead or alive? An analysis of some mutation testing issues. In: 2nd Workshop on Software Testing, Verification, and Analysis. Banff, Canada, July 1988
49. Yu, Y.T., Lau, M.F.: A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. J. Syst. Softw. **79**(5), 577–590 (2006)
50. Zhang, J., Hao, D., Zhang, L., Zhang, L.: To detect abnormal program behaviours via mutation deduction. In: Mutation Testing Workshop, Mutation 2018, to appear