Guiding Dynamic Symbolic Execution toward Unverified Program Executions

Maria Christakis Peter Müller Valentin Wüstholz

Department of Computer Science
ETH Zurich, Switzerland
{maria.christakis, peter.mueller, valentin.wuestholz}@inf.ethz.ch

ABSTRACT

Most techniques to detect program errors, such as testing, code reviews, and static program analysis, do not fully verify all possible executions of a program. They leave executions unverified when they do not check an execution path, check it under certain unjustified assumptions (such as the absence of arithmetic overflow), or fail to verify certain properties.

In this paper, we present a technique to complement partial verification results by automatic test case generation. We annotate programs to reflect which executions have been verified, and under which assumptions. These annotations are then used to guide dynamic symbolic execution toward unverified program executions. Our main contribution is a code instrumentation that causes dynamic symbolic execution to abort tests that lead to verified executions, to prune parts of the search space, and to prioritize tests that lead to unverified executions. We have implemented our technique for the .NET static analyzer Clousot and the dynamic symbolic execution tool Pex. Compared to directly running Pex on the annotated programs without our instrumentation, our approach produces smaller test suites (by up to 19.2%), covers more unverified executions (by up to 7.1%), and reduces testing time (by up to 52.4%).

1. INTRODUCTION

Modern software projects use a variety of techniques to detect program errors, such as testing, code reviews, and static program analysis [30]. In practice, none of these techniques check all possible executions of a program. They often leave entire paths unverified (for instance, when a test suite does not achieve full path coverage), fail to verify certain properties (such as complex assertions), or verify some paths under assumptions (such as the absence of arithmetic overflow) that might not hold on all executions of the path. Making such assumptions is necessary in code reviews to reduce the complexity of the task; it is also common in static program analysis to improve the precision, performance, and modularity of the analysis [12], and because some program features elude static checking [36].


```
1  void Deposit(int amount) {
2    if (amount <= 0 || amount > 50000) {
3      ReviewDeposit(amount);
4    } else {
5      balance = balance + amount;
6      if (balance > 10000) {
7        SuggestInvestment();
8      }
9    }
10   assert balance >= old(balance);
11 }
```

Figure 1: Example illustrating partial verification results. Techniques that assume that the addition on line 5 will not overflow might miss violations of the assertion on line 10. We use the assertion to make the intended behavior explicit; the `old` keyword indicates that an expression is evaluated in the pre-state of the method. `balance` is an integer field declared in the enclosing class. We assume methods `ReviewDeposit` and `SuggestInvestment` to be correct.

Automatic test case generation via dynamic symbolic execution (DSE) [27, 9], also called concolic testing [38], systematically explores a large number of program executions and, thus, effectively detects errors missed by other techniques. However, simply applying DSE in addition to other techniques leads to redundancy when executions covered by DSE have already been verified using other techniques. In this case, the available testing time is wasted on executions that are known to be correct rather than exploring previously-unverified executions. This redundancy is especially problematic when DSE is used to complement static analyzers because static techniques can check a large fraction of all possible program executions and, thus, many or even most of the executions covered by DSE are already verified.

Method `Deposit` in Fig. 1 illustrates this problem. A reviewer or static analyzer that checks the implementation under the assumption that the addition on line 5 will not overflow might miss violations of the assertion on line 10. Applying DSE to the method will try to explore six different paths through the method (there are three paths through the conditionals, each combined with two possible outcomes for the assertion), in addition to all the paths through the called methods `ReviewDeposit` and `SuggestInvestment`. Assuming that these two methods are correct, only one of all these paths reveals an error, namely the path that is taken when

`amount` is between 0 and 50,000, and `balance` is large enough for the addition on line 5 to overflow. All other generated test cases are redundant because they lead to executions that have already been verified. In particular, if the called methods have complex control flow, DSE might not detect the error because it reaches a timeout before generating the only relevant test case.

In this paper, we present a technique to guide DSE toward unverified program executions. Building on our earlier work [11], we use program annotations to make explicit which assertions in a program have already been verified, and under which assumptions. These annotations can be generated automatically by a static analysis [12] or inserted manually, for instance, during a code review. The main contribution of this paper is a code instrumentation of the unit under test that (1) detects redundant test cases early during their execution and aborts them, (2) reduces the search space for DSE by pruning paths that have been previously verified, and (3) prioritizes test cases that cover unverified executions. This instrumentation is based on an efficient static inference that propagates information that characterizes unverified executions higher up in the control flow, where it may prune the search space more effectively.

Our technique works for modular and whole-program verification, and can be used to generate unit or system tests. For concreteness, we present it for modular verification and unit testing. In particular, we have implemented our approach for Microsoft's .NET static checker Clousot [23], a modular static analysis, and the DSE tool Pex [39], a test case generator for unit tests. Our experiments demonstrate that, compared to classical DSE, our approach produces smaller test suites, explores more unverified executions, and reduces testing time.

Outline. We give an overview of our approach in Sect. 2. Sect. 3 explains how we infer the code instrumentation from partial verification results. Our experimental results are presented in Sect. 4. We discuss related work in Sect. 5 and conclude in Sect. 6.

2. APPROACH

In this section, we summarize an annotation language that we have developed in earlier work [11] to express partial verification results, and then illustrate how we use these annotations to guide DSE toward unverified executions. The details of the approach will be explained in the next section.

2.1 Verification annotations

In order to encode partial verification results, we introduce two kinds of annotations: An *`assumed` statement* of the form `assumed` P `as` a expresses that an analysis assumed property P to hold at this point in the code without checking it. The *assumption identifier* a uniquely identifies this statement. In order to record verification results, we use assertions of the form `assert` P `verified` A, which express that property P has been verified under condition A. The *supposition* A is a boolean condition over assumption identifiers, each of which is introduced in an `assumed` statement. It is the conjunction of the identifiers for the assumptions used to verify P, or false if P was not verified. When several verification results are combined (for instance, from a static analysis and a code review), A is the disjunction of the assumptions made during each individual verification. We

```
1  void Deposit(int amount) {
2    var a = true;
3    if (amount <= 0 || 50000 < amount) {
4      assume !a;
5      ReviewDeposit(amount);
6    } else {
7      assumed noOverflowAdd(balance, amount) as a;
8      a = a && noOverflowAdd(balance, amount);
9      assume !a;
10     balance = balance + amount;
11     if (10000 < balance) {
12       SuggestInvestment();
13     }
14   }
15   assume !a || balance >= old(balance);
16   assert balance >= old(balance) verified a;
17 }
```

Figure 2: The instrumented version of the method from Fig. 1. The dark boxes show the annotations generated by the static analyzer. The `assumed` statement makes explicit that the analyzer assumed that the addition on line 10 does not overflow. The `verified` annotation on the assertion on line 16 expresses that the assertion was verified under this (unjustified) assumption. The two annotations are connected via the assumption identifier `a`, which uniquely identifies the `assumed` statement. The light boxes show the instrumentation that we infer from the annotations and that prunes redundant tests.

record verification results for all assertions in the code, including implicit assertions such as a receiver being non-null or an index being within the bounds of an array.

We assume here that a static analyzer records the assumptions it made during the analysis, which assertions it verified, and under which assumptions. We equipped Microsoft's .NET static analyzer Clousot [23] with this functionality [12]. Among other unjustified assumptions, Clousot ignores arithmetic overflow and, thus, misses the potential violation of the assertion on line 10 of Fig. 1[1]. This partial verification result is expressed by the annotations in the dark boxes of Fig. 2 (the light boxes will be discussed below). The `assumed` statement makes explicit that the addition on line 10 was assumed not to overflow (the predicate `noOverflowAdd` can be encoded as equality of an integer and a long-integer addition); the `verified` annotation on the assertion on line 16 expresses that the assertion was verified under this (unjustified) assumption.

The meaning of verification annotations can be defined in terms of assignments and `assume` statements, which makes the annotations easy to support by a wide range of static and dynamic tools. For each assumption identifier, we declare a boolean variable, which is initialized to true. For modular analyses, assumption identifiers are local variables initialized at the beginning of the enclosing method (line 2 in Fig. 2), whereas for whole-program analyses, assumption identifiers

[1]Clousot is modular, that is, reasons about a method call using the method's pre- and postcondition; we assume here that the postconditions of `ReviewDeposit` and `SuggestInvestment` state that `balance` is not decreased.

are global variables, for instance, initialized at the beginning of a main method. A statement `assumed` P `as` a is encoded as

$$a \texttt{ = } a \texttt{ \&\& } P;$$

as illustrated on line 8. That is, the variable a accumulates the assumed properties for each execution of the `assumed` statement. Since assumptions typically depend on the current execution state, this encoding ensures that an assumption is evaluated in the state in which it is made rather than the state in which it is used.

An assertion `assert` P `verified` A is encoded as

$$\texttt{assume } A \Rightarrow P\texttt{;}$$
$$\texttt{assert } P\texttt{;}$$

as illustrated on line 15. The `assume` statement expresses that, if condition A holds, then the asserted property P holds as well, which reflects that P was verified under the supposition A. Consequently, an assertion is unverified if A is false, the assertion is fully verified if A is true, and otherwise, the assertion is partially verified.

2.2 Guiding dynamic symbolic execution

To reduce redundancies with prior analyses of the unit under test, DSE should generate test cases that check each assertion `assert` P `verified` A for the case that the supposition A does not hold, because P has been verified to hold otherwise. DSE can be guided by adding constraints to path conditions, which will then be satisfied by the generated test inputs. The `assume` statement in the encoding of an assertion contributes such a constraint, reflecting that only inputs that violate the supposition A may reveal a violation of the asserted property P. However, these `assume` statements do not effectively guide DSE, as we explain next.

`assume` statements affect DSE in two ways. First, when the execution of a test case encounters an `assume` statement whose condition is false, the execution is aborted. Second, when an execution encounters an `assume` statement, its condition is added to the symbolic path condition, ensuring that subsequent test cases that share the prefix of the execution path up to the `assume` statement will satisfy the condition. Therefore, the effect of an `assume` statement is larger the earlier it occurs in the control flow, because early assumptions may abort test cases earlier and share the prefix with more executions.

However, the `assume` statements we introduce for assertions tend to occur late in the unit under test because assertions often encode postconditions and test oracles. Therefore, these assumptions do not effectively guide DSE toward unverified executions. Our example (Fig. 2) illustrates this problem. First, aborting test cases on line 15, right before the final assertion, saves almost no execution time. Second, the various ways to execute the different branches of the conditionals and the called methods do not share the prefix before the `assume` statement and are, thus, not influenced by it. Consequently, each time DSE determines the next test case, it is likely to choose inputs that do not overflow, that is, to test an execution that has already been verified. In fact, running DSE on the example from Fig. 2 (without lines 4 and 9, which we discuss below) generates exactly as many test cases as if there were no prior verification results.

To address this problem, we propagate constraints that characterize unverified executions higher up in the control flow, where they can be used to effectively prune redundant test cases and to prioritize non-redundant test cases, that is, tests that cover unverified executions.

A test is *redundant* if the supposition of each assertion in its execution holds; in this case, all assertions have been verified. In order to detect redundant tests early, we compute for each program point a sufficient condition for every execution from this program point onward to be verified. If this condition holds, we can abort the execution. We achieve this behavior by instrumenting the unit under test with `assume` statements for the negation of the condition, that is, we assume a necessary condition for the existence of at least one unverified execution from the `assume` statement onward. When the assumption evaluates to false during the execution of a test, it aborts the test and introduces a constraint, which implies that at least one supposition must be violated, for all other test cases with the same prefix.

The example in Fig. 2 has an assertion with supposition `a` at the very end. Consider the program points on lines 4 and 9. At both points, `a` is a sufficient condition for the rest of the execution of `Deposit` to be verified. Since we are interested in test cases the lead to unverified executions, we instrument both program points by assuming the negation, that is, `!a`. With this instrumentation, any test case that enters the outer then-branch is aborted since `a` is always true at this point, which, in particular, prunes the entire exploration of method `ReviewDeposit`. Similarly, any test case that does not lead to an overflow on line 10 is aborted on line 9, which prunes the entire exploration of method `SuggestInvestment`. So, out of all the test cases generated by DSE for the un-instrumented `Deposit` method, only the one that reveals the error remains; all others are either aborted early or pruned.

Since the goal of the instrumentation described so far is to abort or prune redundant test cases, it has to be conservative. Any execution that *may* be unverified cannot be eliminated without potentially missing bugs; hence, we call this instrumentation *may-unverified instrumentation*. If an execution path contains several assertions, which is common because of the implicit assertions for dereferencing, array access, etc., this instrumentation retains any execution in which the supposition of *at least one* of these assertions does not hold.

Intuitively, test cases that violate the supposition of more than one assertion have a higher chance to detect an assertion violation. To prioritize such test cases, we devise a second instrumentation, called *must-unverified instrumentation*: We compute for each program point a sufficient condition for every execution from this program point onward to be *definitely* unverified. If the condition holds, then every execution from the program point onward contains at least one assertion, and the suppositions of *all* assertions in the execution are false.

When the must-unverified condition is violated, it does not necessarily mean that the subsequent execution is verified and, thus, we cannot abort the test case. Therefore, we instrument the program not by assuming the must-unverified condition, but instead with a dedicated `tryfirst` instruction. This instruction interrupts the execution of the test case and instructs DSE to generate new inputs that satisfy the must-unverified condition, that is, inputs that have a higher chance to detect an assertion violation. The interrupted test case is re-generated later, after the execu-

tions that satisfy the must-unverified condition have been explored. This exploration strategy prioritizes test cases that violate all suppositions over those that violate only some.

Suppose that the `Deposit` method in Fig. 2 contained another assertion at the very end that has not been verified, that is, whose supposition is false. In this case, the may-unverified instrumentation yields true for all prior program points since every execution is unverified. In this case, this instrumentation neither aborts nor prunes any test cases. In contrast, the must-unverified instrumentation infers `!a` on line 9. The corresponding `tryfirst` instruction (not shown in Fig. 2) gives priority to executions that lead to an overflow on line 10. However, it does not prune the others since they might detect a violation of the unverified second assertion at the end of the method.

The may-unverified and must-unverified instrumentations have complementary strengths. While the former effectively aborts or prunes redundant tests, the latter prioritizes those tests among the non-redundant ones that are more likely to detect an assertion violation. Therefore, our experiments show the best results for the combination of both.

3. CONDITION INFERENCE

Our may-unverified and must-unverified conditions reflect whether the suppositions of assertions further down in the control flow hold. In that sense, they resemble weakest preconditions [20]: a may-unverified condition is the negation of the weakest condition that implies that all suppositions further down hold; a must-unverified condition is the weakest condition that implies that all suppositions do not hold. However, existing techniques for computing weakest preconditions have shortcomings that make them unsuitable in our context. For instance, weakest precondition calculi [33] require loop invariants, abstract interpretation [14] may require expensive fixed-point computations in sophisticated abstract domains, predicate abstraction [29, 4] may require numerous invocations of a theorem prover for deriving boolean programs [5] from the original program, and symbolic execution [31] struggles with path explosion, for instance, in the presence of input-dependent loops.

In this section, we present two efficient instrumentations that *approximate* the may-unverified and must-unverified conditions of a unit under test. For this purpose, we syntactically compute a non-deterministic abstraction of the unit under test. This abstraction is sound, that is, each execution of the concrete program is included in the set of executions of the abstract program. Therefore, a condition that guarantees that all suppositions hold (or are violated) in the abstract program provides the same guarantee for the concrete program. The may-unverified and must-unverified conditions for the abstract program can be computed efficiently using abstract interpretation, and can then be used to instrument the concrete program.

3.1 Abstraction

We abstract a concrete program to a boolean program [5], where all boolean variables are assumption identifiers. In the abstract program, all expressions that do not include assumption identifiers are replaced by non-deterministically chosen values, which, in particular, replaces conditional control flow by non-determinism. Moreover, the abstraction removes assertions that have been fully verified, that is, where the supposition is the literal `true`. (Note that we consider the supposition to be the literal `true` when `true` is a disjunct of the supposition.)

For simplicity, we present the abstraction for a *concrete* programming language with the statements: `assumed` statements, assertions, method calls, conditionals, loops, and assignments. Besides conditional statements and loops with non-deterministic guards, the *abstract* language provides the following statements:

- initialization of assumption identifiers: `var a := true`,
- updates to assumption identifiers: `a := a && *`, where `*` denotes a non-deterministic (boolean) value,
- assertions: `assert * verified` A, where $A \not\equiv$ `true`, and
- method calls: `call` M_f, where M_f is a fully-qualified method name and the receiver and arguments have been abstracted away.

Note that we desugar `assumed` statements into initializations and updates of assumption identifiers, which allows us to treat modular and whole-program analyses uniformly even though they require a different encoding of `assumed` statements (Sect. 2.1).

To abstract a program, we recursively apply the following transformations to its statements. These transformations can be considered as an application of predicate abstraction [29], which uses the assumption identifiers as predicates and does not rely on a theorem prover to derive the boolean program:

- an assumption `assumed` P `as a` is rewritten to an assumption identifier initialization `var a := true` (at the appropriate program point, as discussed above) and an update `a := a && *`,
- an assertion `assert` P `verified` A is transformed into `assert * verified` A, if A is not the literal `true` (and omitted otherwise),
- a method call $r.M$`(...)` is rewritten to `call` M_f, where M_f is the fully-qualified name of M,
- a conditional statement `if (`b`)` S_0 `else` S_1 is rewritten to `if (*)` S_0' `else` S_1', where S_0' and S_1' are the results of recursively rewriting the statements S_0 and S_1, respectively,
- a loop `while (`b`)` S is rewritten to `while (*)` S', where S' is the result of recursively rewriting statement S, and
- assignments are omitted.

Fig. 3 shows the abstraction of method `Deposit` from Fig. 2. The gray boxes (light and dark) show the inferred may-unverified conditions, as we explain next.

3.2 May-unverified conditions

A may-unverified condition expresses that some execution from the current program point onward may be unverified. We compute this condition for each program point in two steps. First, we compute the weakest condition at the corresponding program point in the abstract program that implies that all executions are verified. Since the set of executions of the abstract program subsumes the set of concrete executions, this condition also implies that all concrete executions are verified (although for the concrete execution, the computed condition is not necessarily the weakest such condition). Second, we negate the computed condition to obtain a may-unverified condition.

Inference.

To compute the weakest condition that implies that all

```
1  method Deposit() {
2    {true}
3    var a := true;
4    {true}
5    if (*) {
6      {!a}
7      call Account.ReviewDeposit;
8      {!a}
9    } else {
10     {true}
11     a := a && *;
12     {!a}
13     if (*) {
14       {!a}
15       call Account.SuggestInvestment;
16       {!a}
17     }
18     {!a}
19   }
20   {!a}
21   assert * verified a;
22   {false}
23 }
```

Figure 3: The abstraction of method `Deposit` from Fig. 2. The gray boxes (light and dark) show the inferred may-unverified conditions. The conditions that are used for the may-unverified instrumentation are shown in dark gray boxes.

executions from a program point onward are verified, we define a predicate transformer WP on abstract programs. If $WP(S, R)$ holds in a state, then the supposition of each assertion in each execution of statement S from that state holds and, if the execution terminates, R holds in the final state. Since we focus on modular verification techniques in this paper, we assume here that calls are encoded via their pre- and postcondition [34] and, thus, do not occur in the abstract program. Defining an inter-procedural WP is of course also possible. Thus, we define WP as follows:

- $WP(\texttt{assert * verified}\ A, R) \equiv A \wedge R$,
- $WP(\texttt{a := true}, R) \equiv R[\texttt{a := true}]$, where $R[\texttt{a := true}]$ denotes the substitution of `a` by true in R, and
- $WP(\texttt{a := a \&\& *}, R) \equiv \forall b :: R[\texttt{a} := b]$, where b is a fresh boolean variable. The quantifier in this predicate can be replaced by the conjunction of the two possible instantiations.

The semantics of sequential composition, conditionals, and loops is standard [20]. In our implementation, we use backward abstract interpretation to compute the weakest precondition for each program point in terms of a set of cubes (that is, conjunctions of assumption identifiers or their negations). In the presence of loops or recursion, we use a fixed-point computation.

For every program point of the abstract program, the may-unverified condition is the negation of the weakest precondition at that program point

$$\textsc{May}(S) \equiv \neg WP(S, true)$$

where S denotes the code fragment after the program point.

The gray boxes in Fig. 3 show the may-unverified conditions at each program point (assuming `ReviewDeposit` and `SuggestInvestment` have no preconditions). In the example, the may-unverified inference propagates meaningful information only up until the non-deterministic update is reached, which corresponds to the `assumed` statement. Specifically, on line 10, we infer true because the abstraction loses the information that would be needed to compute a stronger may-unverified condition. So, in return for an efficient condition inference, we miss some opportunities for aborting and pruning redundant tests.

Instrumentation.

Since each execution of the concrete program corresponds to an execution of the abstract program, we can instrument the concrete program by adding an `assume` C statement at each program point, where C is the may-unverified condition at the corresponding program point in the abstract program. As we explained in Sect. 2.2, these statements will abort redundant test cases and contribute constraints that guide DSE toward unverified executions.

To avoid redundant constraints that would slow down DSE, we omit `assume` statements when the may-unverified condition is trivially true or not different from the condition at the previous program point, as well as the `assume false` statement at the end of the unit under test. Therefore, out of all the conditions inferred for the example in Fig. 3, we use only the ones on lines 6 and 12 to instrument the program, which leads to the assumptions on lines 4 and 9 of Fig. 2 and guides DSE as described in Sect. 2.2.

3.3 Must-unverified conditions

A must-unverified condition expresses that (1) each execution from the program point onward contains at least one assertion and (2) on each execution, the supposition of each assertion evaluates to false. We can compute must-unverified conditions on the abstract program because the conditions for the abstract program provide guarantees for each abstract execution, and each concrete execution corresponds to an abstract execution.

Inference.

We infer the two properties that are entailed by a must-unverified condition separately via two predicate transformers $\textsc{Must}_{assert}$ and $\textsc{Must}_{all}$. If $\textsc{Must}_{assert}(S, R)$ holds in a state, then each execution of statement S from that state encounters at least one assertion or terminates in a state in which R holds. If $\textsc{Must}_{all}(S, R)$ holds in a state, then the supposition of each assertion in each execution of statement S from that state does not hold and, if S terminates, R holds. Both transformers yield the weakest condition that has these properties. Consequently, we obtain the weakest must-unverified condition for an abstract statement S as follows:

$$\textsc{Must}(S) \equiv \textsc{Must}_{assert}(S, false) \wedge \textsc{Must}_{all}(S, true)$$

$\textsc{Must}_{assert}$ and $\textsc{Must}_{all}$ are defined analogously to WP (see Sect. 3.2), except for the treatment of assertions:

$$\textsc{Must}_{assert}(\texttt{assert * verified}\ A, R) \equiv true$$

$$\textsc{Must}_{all}(\texttt{assert * verified}\ A, R) \equiv \neg A \wedge R$$

```
1  method Deposit() {
2    {false}
3    var a := true;
4    {!a}
5    if (*) {
6      {!a}
7      call Account.ReviewDeposit;
8      {!a}
9    } else {
10     {!a}
11     a := a && *;
12     {!a}
13     if (*) {
14       {!a}
15       call Account.SuggestInvestment;
16       {!a}
17     }
18     {!a}
19   }
20   {!a}
21   assert * verified a;
22   {true}
23   assert * verified false;
24   {false}
25 }
```

Figure 4: The abstraction of a variant of method `Deposit` from Fig. 2 that contains an additional unverified assertion at the end of the method (see Sect. 2.2). The gray boxes show the inferred must-unverified conditions. The conditions that are used for the must-unverified instrumentation are shown in dark gray boxes.

The definition for MUST_{assert} expresses that, at a program point before an assertion, property (1) holds, that is, the remaining execution (from that point on) contains at least one assertion. The definition for MUST_{all} expresses that the supposition A must evaluate to false, and that R must hold to ensure that the suppositions of subsequent assertions do not hold either.

Fig. 4 shows the abstraction of a variant of `Deposit` from Fig. 2 that contains an additional unverified assertion at the end of the method (see Sect. 2.2). The gray boxes show the inferred must-unverified conditions, as we explain next. Compared to the may-unverified conditions, the must-unverified conditions are stronger, that is, information is usually propagated further up in the control flow. Whereas the unverified assertion at the end of this example causes the may-unverified conditions to be trivially true, the must-unverified inference obtains conditions that can be used to prioritize test cases.

Instrumentation.

To prioritize tests that satisfy their must-unverified conditions, we instrument the concrete program with `tryfirst` C statements, where C is the must-unverified condition at the corresponding program point in the abstract program. This statement causes DSE to prefer test inputs that satisfy condition C. More specifically, when a `tryfirst` C statement is executed for the first time, it adds C to the path condition to force DSE to generate inputs that satisfy condition C. Note however, that unlike the constraints added by `assume` statements, this constraint may be dropped by the DSE to also explore executions where the condition is violated. If during this first execution of the statement condition C is violated, then the test case is interrupted and will be re-generated later when condition C can no longer be satisfied. So the `tryfirst` statement influences the *order* in which test cases are generated, but never aborts or prunes tests. Nevertheless, the order is important because DSE is typically applied until certain limits (for instance, on the overall testing time or the number of test cases) are reached. Therefore, exploring non-redundant test cases early increases effectiveness.

To avoid wasting time on interrupting tests that will be re-generated later, our implementation enforces an upper bound on the number of interrupts that are allowed per unit under test. When this upper bound is exceeded, all remaining `tryfirst` statements have no effect.

As illustrated by lines 4, 6, 8, and 10 in Fig. 4, the must-unverified condition at some program points evaluates to false for all executions. Instrumenting these program points would lead to useless interruption and re-generation of test cases. To detect such cases, we apply constant propagation and do not instrument program points for which the must-unverified conditions are trivially true or false. Moreover, we also omit the instrumentation for conditions that are not different from the condition at the previous program point. Therefore, out of all the conditions inferred for the example in Fig. 4, we use only the ones on lines 12 and 20 to instrument the program, which prioritize test cases that lead to an arithmetic overflow on line 10, as discussed in Sect. 2.2.

3.4 Combined instrumentation

As we explained in Sect. 2.2, the may-unverified instrumentation aborts and prunes redundant tests, while the must-unverified instrumentation prioritizes test cases that are more likely to detect an assertion violation. One can, therefore, combine both instrumentations such that DSE (1) attempts to first explore program executions that must be unverified, and (2) falls back on executions that may be unverified when the former is no longer feasible.

The combined instrumentation includes both the `assume` statements from the may-unverified instrumentation and the `tryfirst` statements from the must-unverified instrumentation. The `tryfirst` statement comes first. Whenever we can determine that the must-unverified and may-unverified conditions at a particular program point are equivalent, we omit the `tryfirst` statement, because any interrupted and re-generated test case would be aborted by the subsequent `assume` statement anyway.

4. EXPERIMENTS

In this section, we give an overview of our implementation and present our experimental results. In particular, we show that, compared to dynamic symbolic execution alone, our technique produces smaller test suites, covers more unverified executions, and reduces testing time. We also show which of our instrumentations—may-unverified, must-unverified, or their combination—is the most effective.

4.1 Implementation

We have implemented our technique for the .NET static analyzer Clousot [23] and the dynamic symbolic execution tool Pex [39]. Our tool chain consists of four subsequent stages: (1) static analysis and verification-annotation instrumentation, (2) may-unverified and must-unverified instrumentation, (3) runtime checking, and (4) dynamic symbolic execution.

The first stage runs Clousot on a given .NET program, which contains code and optionally specifications expressed in Code Contracts [22], and instruments the sources of unsoundness and verification results of the analyzer using our verification annotations. For this purpose, we have implemented a wrapper around Clousot, which we call *Inspector-Clousot*, that uses the debug output emitted during the static analysis to instrument the program (at the binary level). Note that Clousot performs a modular analysis, and thus, the verification annotations are local to the containing methods.

The second stage of the tool chain adds the may-unverified, must-unverified instrumentation, or their combination to the annotated program.

In the third stage, we run the existing Code Contracts binary rewriter to transform any Code Contracts specifications into runtime checks. We then run a second rewriter, which we call *Runtime-Checking-Rewriter*, that transforms all the `assumed` statements and assertions of the annotated program into assignments and assumptions, as described in Sect. 2.1.

In the final stage, we run Pex on the instrumented code.

4.2 Experimental evaluation

In the rest of this section, we describe the setup for the evaluation of our technique and present experiments that evaluate its benefits.

Setup.

For our experiments, we used 101 methods from nine open-source C# projects and from solutions to 13 programming tasks on the Rosetta Code repository. We selected only methods for which Pex can automatically produce more than one test case (that is, Pex does not require user-provided factories) and at least one successful test case (that is, Pex generates non-trivial inputs that, for instance, pass input validation that might be performed by the method).

In Clousot, we enabled all checks, set the warning level to the maximum, and disabled all inference options. In Pex, we set the maximum number of branches, conditions, and execution tree nodes to 100,000, and the maximum number of concrete runs to 30.

In our experiments, we allowed up to 4 test interrupts per method under test when these are caused by `tryfirst` statements (see Sect. 3.3). We experimented with different such bounds (1, 2, 4, and 8) on 25 methods from the suite of 101 methods. We found that, for an upper bound of 4 for the number of allowed interrupts per method, dynamic symbolic execution strikes a good balance between testing time and the number of detected bugs.

Performance of static analysis and instrumentation.

On average, Clousot analyzes each method from our suite in 1.9 seconds. The may-unverified and must-unverified instrumentations are very efficient. On average, they need 22 milliseconds per method when combined.

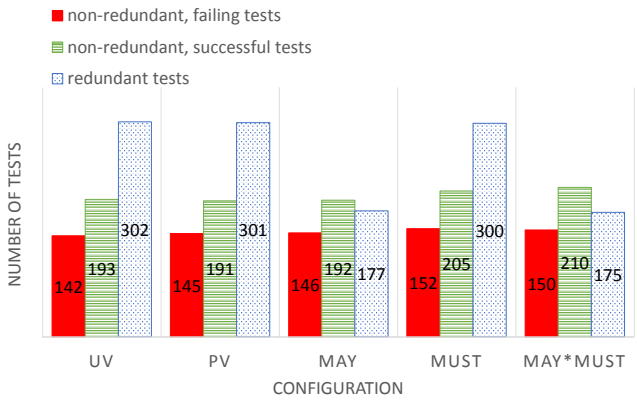

Figure 5: The tests generated by each configuration, categorized as non-redundant and failing, or non-redundant and successful, or redundant tests.

Configurations.

To evaluate our technique, we use the following configurations:

- UV: *unverified* code.
  Stages 1 and 2 of the tool chain are not run.
- PV: *partially-verified* code.
  Stage 2 of the tool chain is not run.
- MAY: partially-verified code, instrumented with *may*-unverified conditions.
  All stages of the tool chain are run. Stage 2 adds only the may-unverified instrumentation.
- $MUST$: partially-verified code, instrumented with *must*-unverified conditions.
  All stages of the tool chain are run. Stage 2 adds only the must-unverified instrumentation.
- $MAY \times MUST$: partially-verified code, instrumented with *may*-unverified and *must*-unverified conditions.
  All stages of the tool chain are run. Stage 2 adds the combined may-unverified and must-unverified instrumentation.

For our experiments, we use configuration PV as the baseline to highlight the benefits of the main contributions of this paper, that is, of the may-unverified and must-unverified inference. However, note that the results of dynamic symbolic execution alone, that is, of UV, do not significantly differ from those of PV in terms of the total number of tests and the number of non-redundant tests generated for the 101 methods.

This is illustrated by Fig. 5, which shows the tests that each configuration generated for the 101 methods, categorized as non-redundant and failing, or non-redundant and successful, or redundant tests. To determine the redundant tests generated by configurations UV, PV, and $MUST$, we ran all tests generated by these configurations against the 101 methods, after having instrumented the methods with the may-unverified conditions. We then counted how many of these tests were aborted. Note that the figure does not include tests that are interrupted because a condition in a `tryfirst` statement is violated (since these tests are regenerated—and counted—later). Furthermore, tests that

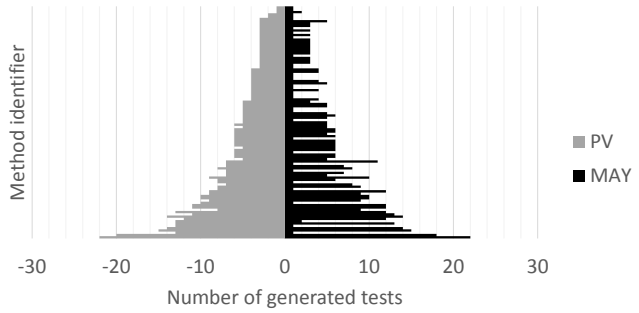

Figure 6: Total number of tests generated for the 101 methods by configurations PV and MAY.

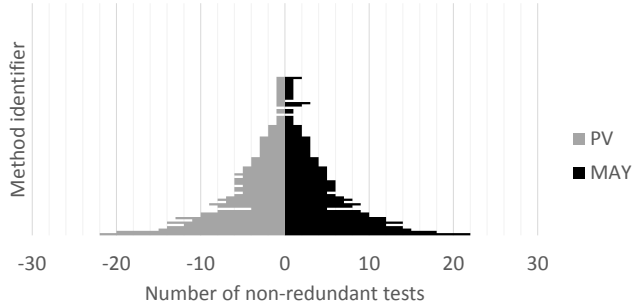

Figure 7: Number of non-redundant tests generated for the 101 methods by configurations PV and MAY.

terminate on exceptions that are explicitly thrown by the method under test, for instance, for parameter validation, are not considered failing.

Smaller test suites.

The may-unverified instrumentation causes DSE to abort tests leading to verified executions. By aborting these tests, our technique prunes the verified parts of the search space that would be explored only if these tests were not aborted. As a result, DSE generates smaller test suites.

This is shown in Fig. 6. In comparison to PV, the total number of tests generated for the 101 methods by MAY is significantly smaller. Note that Fig. 6 includes all generated tests, even those that are aborted. However, for certain methods, configuration MAY generates more tests than PV. This is the case when pruning verified parts of the search space guides dynamic symbolic execution toward executions that happen to be easier to cover within the exploration bounds of Pex (for instance, maximum number of branches or constraint solver timeouts).

Fig. 5 shows that, in total, MAY generates 19.2% fewer tests and $MAY \times MUST$ generates 16.1% fewer tests than PV. The differences in the total number of tests for configurations without the may-unverified instrumentation are minor.

More unverified executions.

Even though configuration MAY generates smaller test suites in comparison to PV, it does not generate fewer non-redundant tests, as shown in Fig. 7. In other words, MAY generates at least as many non-redundant tests as PV, thus covering at least as many unverified executions. Fig. 5 shows

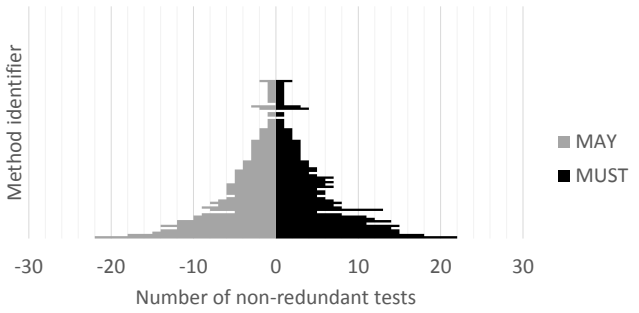

Figure 8: Number of non-redundant tests generated for the 101 methods by configurations MAY and $MUST$.

that MAY generates two additional non-redundant tests in comparison to PV.

The must-unverified instrumentation causes dynamic symbolic execution to prioritize test inputs that lead to unverified executions. In comparison to the may-unverified conditions, the must-unverified conditions are stronger and their instrumentation is usually added further up in the control flow. As a result, this instrumentation can guide dynamic symbolic execution to cover unverified executions earlier and may allow it to generate more tests for such executions within the exploration bounds of Pex. This is shown in Fig. 8. Fig. 5 shows that configuration $MUST$ generates 5.6% more non-redundant tests than MAY and 6.3% more than PV. By generating more such tests, we increase the chances of producing more failing tests. In fact, $MUST$ generates 4.1% more failing tests than MAY and 4.8% more than PV.

$MUST$ typically generates more non-redundant tests for methods in which Clousot detects errors, that is, for methods with unverified assertions. In such methods, the may-unverified instrumentation is added only after the unverified assertions in the control flow (if the conditions are non-trivial), thus failing to guide dynamic symbolic execution toward unverified executions early on, as discussed in Sect. 2.2.

Shorter testing time.

We now compare the testing time of the different configurations. For this experiment, we considered only methods for which all configurations generated the same number of non-redundant tests. This is to ensure a fair comparison; for these methods, all configurations achieved the same coverage of unverified executions. This experiment involved 72 out of the 101 methods, and the time it took for each configuration to test these methods is shown in Fig. 9. As expected, pruning verified parts of the search space with the may-unverified instrumentation is very effective. In particular, configuration MAY is 51.7% faster and configuration $MAY \times MUST$ is 52.4% faster than PV. Note that Fig. 9 does not include the time of the static analysis for two reasons. First, Clousot is just one way of obtaining verification results. Second, the goal of our work is to efficiently complement verification results with test case generation; so obtaining the verification results is a separate step. Recall that the overhead of the instrumentation is negligible. The differences in performance between the configurations without the may-unverified instrumentation are significantly less

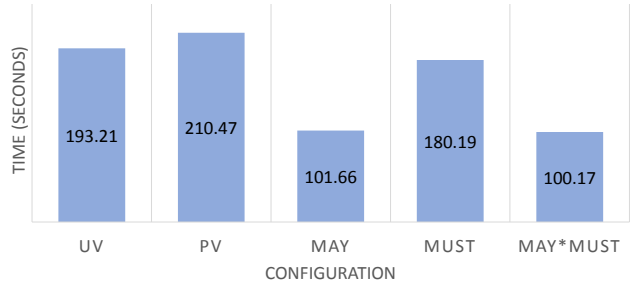

Figure 9: Testing time for each configuration. We only considered methods for which all configurations generated the same number of non-redundant tests.

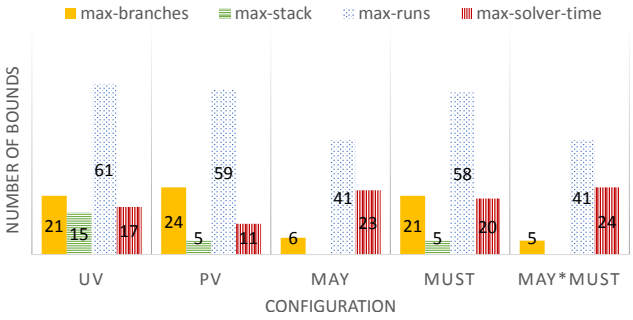

Figure 10: The exploration bounds reached by each configuration, grouped into max-branches, max-stack, max-runs, and max-solver-time.

pronounced.

Even though MAY is overall much faster than PV, there were methods for which the testing time for MAY was longer in comparison to PV. This is the case when constraint solving becomes more difficult due to the inferred conditions. In particular, it might take longer for the constraint solver to prove that an inferred condition at a certain program point does not hold.

Fewer exploration bounds reached.

During its exploration, dynamic symbolic execution may reach bounds that prevent it from covering certain, possibly failing, execution paths. Fig. 10 shows the exploration bounds in Pex that were reached by each configuration when testing all 101 methods. There are four kinds of reached bounds:

- *max-branches*: maximum number of branches that may be taken along a single execution path;
- *max-stack*: maximum size of the stack, in number of active call frames, at any time during a single execution path;
- *max-runs*: maximum number of runs that will be tried during an exploration (each run uses different inputs but not every run results in the generation of a new test case);
- *max-solver-time*: maximum number of seconds that the constraint solver has to find inputs that will cause a different execution path to be taken.

As shown in the figure, configurations MAY, $MUST$, and $MAY \times MUST$ reach the max-solver-time bound more often than PV. This is because our instrumentation introduces additional conjuncts in the path conditions, occasionally making constraint solving harder. Nevertheless, configurations MAY and $MAY \times MUST$ overall reach significantly fewer bounds than PV (for instance, the max-stack bound is never reached) by pruning verified parts of the search space. This helps in alleviating an inherent limitation of symbolic execution by building on results from tools that do not suffer from the same limitation.

Winner configuration.

As shown in Fig. 5, configuration $MAY \times MUST$ generates the second smallest test suite containing the largest number of non-redundant tests and the smallest number of redundant tests. This is achieved in the shortest amount of testing time for methods with the same coverage of unverified executions across all configurations (Fig. 9) and by reaching the smallest number of exploration bounds (Fig. 10).

Therefore, $MAY \times MUST$ effectively combines the benefits of both the may-unverified and must-unverified instrumentation to prune parts of the search space that lead only to verified executions as well as to identify and prefer test inputs that lead to unverified executions as soon as possible.

Soundness bugs in Clousot.

During our experiments, we realized that our verification annotations can also be used to systematically test for soundness issues in static analyzers [18]. This is achieved as follows. Given a piece of code, imagine that configuration UV generates a number of failing tests. Now, we instrument the code with the known unsound assumptions made by a static analyzer and its verification results (stage 1 of the tool chain). We detect a soundness issue if, when running the failing tests against the instrumented code, at least one failing test runs through an assertion `assert` P `verified` A where $A \not\Rightarrow P$. A soundness issue could either be caused by accidental unsoundness (that is, bugs in the implementation of the analyzer) or by bugs in *Inspector-Clousot* (for instance, missing a source of deliberate unsoundness).

In this way, we found the following three bugs in the implementation of Clousot: (1) unsigned integral types are not always treated correctly, (2) the size of each dimension in a multi-dimensional array is not checked to be non-negative upon construction of the array, and (3) arithmetic overflow is ignored in modulo operations (for instance, `MinValue % -1`). We reported these three bugs to the main developer of Clousot, Francesco Logozzo, who confirmed all of them. The latter two bugs have already been fixed in the latest version of the tool[2].

Threats to validity.

We identified the following threats to the validity of our experiments:

- *Sample size*: For our experiments, we used 101 methods from nine C# projects and from solutions to 13 programming tasks. We believe that this sample is representative of a large class of C# methods.
- *Static analyzer*: For our experiments, we used a modular (as opposed to whole-program) static analyzer, namely,

[2] `https://github.com/Microsoft/CodeContracts` (revs: `803e34e72061b305c1cde37a886c682129f1ddeb` and `1a3c0fce9f8c761c3c9bb8346291969ed4285cf6`)

Clousot. Moreover, our experimental results depend on the deliberate sources of unsoundness and verification results of this particular analyzer. Note that there are a few sources of unsoundness in Clousot that our tool chain does not capture [12], for instance, about reflection or unmanaged code.

- *Soundly-analyzed methods*: 23 out of the 101 methods contain no `assumed` statements. In case Clousot reports no warning, these methods are fully verified and need not be tested. Other code bases could have a smaller fraction of fully-verified methods, leading to less effective pruning.
- *Failing tests*: The failing tests generated by each configuration do not necessarily reveal bugs in the containing methods. This is inherent to unit testing since methods are tested in isolation rather than in the context of the entire program. However, 50 out of the 101 methods validate their parameters (and for 10 methods no parameter validation was necessary), which suggests that programmers did intend to prevent failures in these methods.

5. RELATED WORK

Many static analyzers that target mainstream programming languages deliberately make unjustified assumptions in order to increase automation, improve performance, and reduce the number of false positives and the annotation overhead for the programmer. Examples of such analyzers are HAVOC [3], Spec# [6], and ESC/Java [24]. Our technique can effectively complement these analyzers by dynamic symbolic execution.

Integration of static analysis and testing.

Various approaches combine static analysis and testing mainly to determine whether an error reported by the static analysis is spurious. Check 'n' Crash [16] is an automated defect detection tool that integrates the ESC/Java static checker with the JCrasher [15] testing tool in order to decide whether errors emitted by the static checker are real bugs. Check 'n' Crash was later integrated with Daikon [21] in the DSD-Crasher tool [17]. DyTa [25] integrates Clousot with Pex to reduce the number of spurious errors compared to static analysis alone, and to perform more efficiently compared to dynamic test generation alone. This is achieved by guiding dynamic symbolic execution toward the errors that Clousot reports. In contrast to our approach, DyTa does not take into account the unjustified assumptions made by Clousot [12]. Consequently, bugs might be missed since execution paths are pruned only based on the reported errors.

YOGI [37] switches between static analysis and dynamic symbolic execution to find bugs, similarly to counterexample-guided abstraction refinement (CEGAR) [13]. Unlike our technique, YOGI relies on a sound static analysis. The SANTE tool [10] also uses a sound value analysis (in combination with program slicing) to prune those execution paths that do not lead to unverified assertions.

A recent approach [19] starts by running a conditional model checker [7] on a program, and then tests those parts of the state space that were not covered by the model checker (for instance, due to timeouts). More specifically, the model checker produces an output condition, which captures the safe states and is used to produce a residual program that can be subsequently tested. Unlike an instrumented program in our technique, the residual program can be structurally very different from the original program. As a result, its construction can take a significant amount of time, as the authors point out. Furthermore, this approach can only characterize assertions as either fully verified or unverified on a given execution path. It is not clear how to apply this approach in a setting with static analysis tools that are not fully sound [36, 12] without reducing its effectiveness.

Dynamic symbolic execution.

Testing and symbolically executing all feasible program paths is not possible in practice. The number of feasible paths can be exponential in the program size, or even infinite in the presence of input-dependent loops.

Existing testing tools based on dynamic symbolic execution alleviate path explosion using search strategies and heuristics, which guide the search toward least-covered parts of the program while pruning the search space. These strategies typically optimize properties such as "deeper paths" (in depth-first search), "less-traveled paths" [35], or "number of new instructions covered" (in breadth-first search). For instance, SAGE [28] uses a generational-search strategy in combination with simple heuristics, such as flip count limits and constraint subsumption. Other industrial-strength tools, like Pex, also use similar techniques. Our technique resembles a search strategy in that it optimizes unverified executions, prunes verified executions, and is guided by verification annotations, instead of properties like the above.

Compositional symbolic execution [26, 1] has been shown to alleviate path explosion. Dynamic state merging [32] and veritesting [2] alleviate path explosion by merging subprogram searches, while RWset [8] prunes searches by dynamically computing variable liveness. By guiding dynamic symbolic execution toward unverified program executions, our technique also alleviates path explosion. In particular, the may-unverified instrumentation causes dynamic symbolic execution to abort tests that lead to verified executions. When aborting these tests, our technique prunes the parts of the search space that would be discovered only if these tests were not aborted. Besides, we could combine our technique with some of these approaches.

6. CONCLUSION

We have presented a technique for complementing partial verification results by automatic test case generation. Our technique causes dynamic symbolic execution to abort tests that lead to verified executions, consequently pruning parts of the search space, and to prioritize tests that lead to unverified executions. It is applicable to any program with verification annotations, either generated automatically by a static analysis or inserted manually, for instance, during a code review. Our work suggests a novel way to combine static analysis and testing in order to maximize software quality, and investigates to what extent static analysis reduces the test effort.

Acknowledgments.

We thank Francesco Logozzo, Mike Barnett, Manuel Fähndrich, and Herman Venter for their valuable help and feedback. We are also grateful to Nikolai Tillman and Jonathan "Peli" de Halleux for sharing the Pex source code with us.

7. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann.

Demand-driven compositional symbolic execution. In *TACAS*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008.

[2] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *ICSE*, pages 1083–1094. ACM, 2014.

[3] T. Ball, B. Hackett, S. K. Lahiri, S. Qadeer, and J. Vanegue. Towards scalable modular checking of user-defined properties. In *VSTTE*, volume 6217 of *LNCS*, pages 1–24. Springer, 2010.

[4] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213. ACM, 2001.

[5] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR–TR–2000–14, Microsoft Research, 2000.

[6] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *CACM*, 54:81–91, 2011.

[7] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: A technique to pass information between verifiers. In *FSE*, pages 57–67. ACM, 2012.

[8] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS*, volume 4963 of *LNCS*, pages 351–366. Springer, 2008.

[9] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, volume 3639 of *LNCS*, pages 2–23. Springer, 2005.

[10] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *TAP*, volume 6706 of *LNCS*, pages 78–83. Springer, 2011.

[11] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.

[12] M. Christakis, P. Müller, and V. Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, volume 8931 of *LNCS*, pages 333–351. Springer, 2015.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[15] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *SPE*, 34:1025–1050, 2004.

[16] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.

[17] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *TOSEM*, 17:1–37, 2008.

[18] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In *NFM*, volume 7226 of *LNCS*, pages 120–125. Springer, 2012.

[19] M. Czech, M.-C. Jakobs, and H. Wehrheim. Just test what you cannot verify! In *FASE*, volume 9033 of *LNCS*, pages 100–114. Springer, 2015.

[20] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18:453–457, 1975.

[21] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, 2007.

[22] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110. ACM, 2010.

[23] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, volume 6528 of *LNCS*, pages 10–30. Springer, 2010.

[24] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.

[25] X. Ge, K. Taneja, T. Xie, and N. Tillmann. DyTa: Dynamic symbolic execution guided with static verification results. In *ICSE*, pages 992–994. ACM, 2011.

[26] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54. ACM, 2007.

[27] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[28] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151—166. The Internet Society, 2008.

[29] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[30] G. J. Holzmann. Mars code. *CACM*, 57:64–73, 2014.

[31] J. C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[32] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, pages 193–204. ACM, 2012.

[33] K. R. M. Leino. Efficient weakest preconditions. *IPL*, 93:281–288, 2005.

[34] K. R. M. Leino. Specification and verification of object-oriented software. Lecture notes of Marktoberdorf International Summer School, 2008.

[35] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *OOPSLA*, pages 19–32. ACM, 2013.

[36] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 58:44–46, 2015.

[37] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The YOGI project: Software property

checking via static analysis and testing. In *TACAS*, volume 5505 of *LNCS*, pages 178–181. Springer, 2009.

[38] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC*, pages 263–272. ACM, 2005.

[39] N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.