

Seeding Contradiction: a fast method for generating full-coverage test suites

Li Huang^{1, 3}, Bertrand Meyer^{1, 2, 3}, Manuel Oriol^{1, 3}

¹Constructor Institute of Technology, Schaffhausen, Switzerland,
<https://institute.constructor.org>.

²Eiffel Software, Santa Barbara, California, <https://eiffel.org>.

³Constructor University, Bremen, Bremen, Germany,
<https://constructor.university>.

Contributing authors: `{li.huang, bm, mo}@constructor.org`;

Abstract

The regression test suite, a key resource for managing program evolution, needs to achieve 100% branch coverage, or very close, to be useful.

Devising a test suite manually is unacceptably tedious. Although many automated methods exist to improve the efficiency of test generation, they are mostly dynamic. The method described in this article, “Seeding Contradiction”, inserts incorrect instructions into every basic block of the program, enabling an SMT-based Hoare-style prover to generate a counterexample for every branch of the program and, from the collection of all such counterexamples, a test suite.

The method is static and works fast. It achieves full coverage, going beyond the usual concept of branch coverage by “unrolling” loops a parameterizable number of times.

Preprint of article to appear in Springer Nature Computer Science, 2025. Revised and extended version of contribution to ICTSS Conference, Bergamo, 2023 [1].

Keywords: Testing, Coverage, Software verification, Eiffel

1 Overview

In the modern theory and practice of software engineering, tests have gained a place of choice among the artifacts of software production, on an equal footing with code. One particularly important rule is that every deployed program should come accompanied with a *regression test suite* achieving high branch coverage and making it possible

to check, after any change to the software, that previous functionality still works: no “regression” has occurred.

Producing a high-coverage regression test suite is a delicate and labor-intensive task. Tools exist (RANDOOP [2], Pex [3], AutoTest [4], Korat [5], KeY [6], jCUTE [7]) but they are typically *dynamic*, meaning that they require numerous executions of the code. The Seeding Contradiction (SC) method and supporting tools presented in this article typically achieve 100% coverage (excluding unreachable code, which they may help detect) and involve no execution of the code, ensuring very fast results.

The principal insight of Seeding Contradiction is to exploit the power of modern program provers, which attempt to generate a counterexample of program correctness. In normal program proving, we hope that the prover will *not* find such a counterexample: a proof follows from the demonstrated *inability* to *disprove* the program’s correctness. Switching the focus from proofs to tests, we may look at counterexamples in a different way: as test cases. We may call this approach *Failed Proofs to Failing Tests* or FP-FT. Previous research (including by some of the present authors) has exploited FP-FT in various ways [8] [9] [10]. Seeding Contradiction extends FP-FT to a new goal: generating a full-coverage test suite, by applying FP-FT to *seeded* versions of the program in which a branch has on purpose been made incorrect. For every such variant, the prover generates a counterexample exercising the corresponding branch. Combining the result for all branches yields a high-coverage test suite. In fact coverage is normally 100%, with the following provisions:

- Some branches may be unreachable. Then by definition no test could cover them; the tool may help identify such cases. (Terminology: we will use the term **exhaustive coverage** to mean 100% coverage of reachable branches.)
- Limitations of the prover may prevent reaching 100%. In our examples so far such cases do not arise.

The method involves no execution of the code and on examples tried so far produces a test suite much faster than dynamic techniques (section 5).

A recent and promising extension of the approach makes it possible to go beyond branch coverage, which in its standard sense considers at most one execution of every loop (treating loops, in practice, as if they were mere conditional instructions). By *unrolling* loops to a parameterizable depth, it generates tests that execute each loop body the corresponding number of times, not just zero or one time as in branch coverage. On examples covered so far unrolling has proved to be of practical interest, uncovering regression bugs not found otherwise, sometimes only after an unrolling depth of 9 or more.

The current setup involves the AutoProof [11] [12] verification framework for contract-equipped Eiffel [13] code, relying internally on the Boogie proof system [14] [15] and the Z3 SMT solver [16]. It is generalizable to other Hoare-style program verification approaches.

The discussion is organized as follows. Section 2 presents the approach by considering a small example. Section 3 examines the theoretical correctness of that approach. Section 4 describes the extent to which we have applied it so far, and section 5 assesses the results. Section 6 discusses limitations of the current state of the work and threats

to validity of the evaluation results. Section 7 reviews related work and section 8 presents conclusions and future work.

2 The method

A simple code example will illustrate the essential idea behind Seeded Contradiction.

2.1 Falsifying a code block

Consider a small routine consisting of a single conditional instruction:

```
simple (a: INTEGER)
do
  if a > 0 then x := 1 else x := 2 end
end
```

where x is an integer attribute of the enclosing class. In a Design-by-Contract approach intended to achieve correctness by construction, the routine might include the following postcondition part (with \implies denoting implication):

```
ensure
  a > 0  $\implies$  x = 1
  a  $\leq$  0  $\implies$  x = 2
```

With or without the postcondition, how can we obtain a regression test suite that will exercise both branches?

Various techniques exist, discussed in section 7 and mostly requiring execution of the code. The Seeding Contradiction technique is, as noted, static (it does not involve executing the code); it assumes that we have a toolset for proving program correctness. Specifically, we rely on the AutoProof environment [11] [12], with a tool stack presented in Fig. 1, in which the Boogie prover is itself based on an SMT solver, currently Z3. A characteristic of this style of proof is that it relies on a *disproof* of the *opposite* property: the SMT solver tries to construct at least one counterexample, violating the desired result. If it cannot find one, the proof is successful.

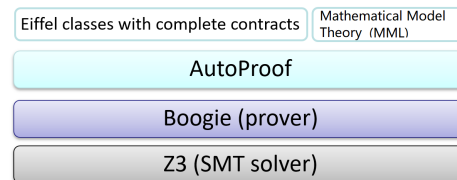


Fig. 1: AutoProof tool stack

In this work, as in previous articles using the general FP-FT approach [9] [10], we are interested in a proof that actually fails: then the counterexample can be useful on its own, yielding a directly usable test.

In contrast with the earlier FP-FT work, the proof that will fail is not a proof of the actual program but of a modified version, into which we have inserted (“seeded”) incorrect instructions. In the example, we change the first branch, so that the routine now reads

```

simple (a: INTEGER)
do
  if a > 0 then
    check False end -- This is the added instruction
    x := 1 -- The rest is unchanged.
  else
    x := 2
  end
end
end

```

A “`check C end`” instruction (`assert C` in some other notations [17]) states that the programmer expects condition `C` to hold at the corresponding program point. Specifically, its semantics is the following, from both a dynamic perspective (what happens if it gets executed) and a static, proof-oriented perspective:

- From a dynamic viewpoint, executing the instruction means: if condition `C` has value `True` at that point, the `check` instruction has no effect other than evaluating `C`; if `C` evaluates to `False` and the programmer has enabled run-time assertion monitoring, as possible in EiffelStudio, execution produces a violated-assertion exception, usually implying that it terminates abnormally.
- In the present discussion’s static approach, the goal is to prove the program correct. The semantics of the `check` instruction is that it is correct if and only if the condition `C` always has value `True` at the given program point. If the prover cannot establish that property, the proof fails.

In a general FP-FT approach, the key property is that in the static view, if the proof fails, an SMT-based prover will generate a **counterexample**. In the Seeding Contradiction approach, `C` is `False`: the proof *always* fails and we get a counterexample exercising the corresponding branch — exactly what we need if, as part of a regression test suite, we want a test exercising the given branch.

For the `simple` code seeded with a `check False end` (called a *trap property* in [18, 19]), such a counterexample will, by construction, lead to execution of the first branch (`a > 0`) of the conditional. If we have an efficient mechanism to turn counterexamples into tests, as described in earlier work [9, 10], we can get, out of this counterexample, a test of the original program which exercises the first branch of the conditional.

Such a generated test enjoys several interesting properties:

- It can be produced even in the absence of a formal specification (contract elements such as the postcondition above).
- Unless the enclosing code (here the routine `simple`) is unreachable, the test can be produced whether the program is correct or incorrect.
- If the program is correct, the test will pass and is useful as a regression test (which may fail in a later revision of the program that introduces a bug).
- Generating it does not require any execution.
- That generation process is fast in practice (section 5).

The next sections will show how to generalize this idea to produce such tests not only for one branch as here but for *all* branches of the program, as needed to obtain an exhaustive-coverage regression test suite.

2.2 Block variables

To generalize the approach, the following terminology is useful. So far it has been convenient to talk informally of “branches”, but the more precise concept is **basic block**, defined in the testing and compilation literature as a sequence of instructions not containing conditionals or loops. (This definition is for a structured program with no branching instructions. In a more general approach, a basic block is any process node — as opposed to decision nodes — in the program’s flowchart.) “Block” as used below is an abbreviation for “basic block”.

The method illustrated on the `simple` example generates a test guaranteed to exercise a specific block of a correct program: seed the program by adding to the chosen block one `check False end` instruction. Then, as seen in the example, we run the prover and apply the FP-FT scheme: since the program is now incorrect, the proof fails and the prover generates a counterexample, which we turn into a runnable test guaranteed to exercise the given block in the original program.

To generalize this approach so that it will generate a test suite exercising all blocks, a straightforward idea is “*Multiple Seeded Programs*” (MSP): generate such a seeded program for each of its blocks in turn; then run the prover on every such program, in each case producing a counterexample and generating a test from it. Subject to conditions in section 3 below, the MSP approach is correct, in the sense that together the generated tests exercise all reachable blocks. It is, however, impractical: for a single original program, we would need to generate a possibly very large number of seeded programs, and run every one of them through the prover.

To obtain a realistic process, we can instead generate a single seeded program, designed to produce the same counterexamples as would all the MSP-generated programs taken together. A helpful property of a good counterexample-based prover is that it can deal with a program containing several faults and generate a set of counterexamples, each addressing one of the faults. In the example above, we can submit to the prover a single seeded program of the form

```
simple (a: INTEGER)
  do
    if a > 0 then
      check False end
      x := 1 -- Instruction 1
    else
      check False end
      x := 2 -- Instruction 2
    end
  end
end
```

which will produce two counterexamples, one for each branch. We call this approach “RSSP” (Repeatedly Seeded Single Program). With AutoProof, the FP-FT tools generate tests with `a = 1` and `a = 0`. (More precisely, the prover initially generates larger

and less intuitive values, but a minimization technique described in earlier work [10] produces 1 and 0.)

This approach does not suffice for more complex examples. Assume that after the conditional instruction the routine `simple` includes another conditional:

```

-- This code comes after the above conditional (Instructions 1-2)
if a2 > a then
  x := 3 -- Instruction 3
else
  x := 4 -- Instruction 4
end

```

With the program seeded as above, even if we insert a `check False end` into each of the two new blocks (before Instructions 3 and 4), we will get tests covering only two cases (1-4, 2-4), not four (1-3, 1-4, 2-3, 2-4) as needed. These two tests, $a = 1$ and $a = 0$, fail to cover Instruction 3. The reason is that the prover does not generate specific tests for the branches of the second conditional (3-4) since it correctly determines that they are unreachable as both branches of the first conditional (1-2) now include a `check False end`. They were, however, both reachable in the original! The test suite fails to achieve exhaustive coverage.

The solution to this “*Seeded Unreachability*” issue is to make the `check` themselves conditional. In the seeded program, for every routine under processing, such as `simple`, we may number every basic block, from 1 to some N , and add to the routine an argument bn (for “block number”) with an associated precondition

```

require
  bn ≥ 0 -- See below why 0 and not 1.
  bn ≤ N

```

To avoid changing the routine’s interface (as the addition of an argument implies), we will instead make bn a local variable and add an initial instruction that assigns to bn , non-deterministically, a value between 0 and N . Either way, we now use, as seeded instructions, no longer just `check False end` but

```

if bn = i then check False end end

```

where i is the number assigned to the block. In the example, the fully seeded routine body for the extended version of `simple` with two conditionals, is (choosing the option of making bn a local variable rather than an argument):

```

bn := “Value chosen non-deterministically between 0 and N”
if a > 0 then
  if bn = 1 then check False end end
  x := 1 -- Instruction 1
else
  if bn = 2 then check False end end
  x := 2 -- Instruction 2
end
if a2 > a then
  if bn = 3 then check False end end
  x := 3 -- Instruction 3

```

```

else
  if  $bn = 4$  then check False end end
  x := 4 -- Instruction 4
end

```

As in the previous attempt, there are four incorrect `check False end` instructions, but all are now reachable for bn values ranging from 1 to 4. The prover generates counterexamples exercising all the paths of the original program (with appropriately generated values for its original variables). In this case there is only one relevant variable, a ; AutoProof’s prover generates, for the pair $[bn, a]$, the test values $[1, 1]$, $[2, 0]$, $[3, -1]$, $[4, 0]$. These four tests provide 100% branch coverage for the program and can serve as a regression test suite. We call this technique **Conditional Seeding**; it addresses the Seeded Unreachability issue.

As noted above, we accept for bn not only values between 1 and N (the number of basic blocks) but also 0. This convention has no bearing on test generation and coverage but ensures that the behavior of the original program remains possible in the seeded version: for $bn = 0$, none of the seeded `check False end` will execute, so the program behaves exactly as the original. If the original was correct, the prover will not generate any counterexample for that value.

2.3 SCU: seeding contradictions with loop unrolling

The Seeding Contradiction strategy, as noted, achieves exhaustive coverage, in the sense of exhaustive *branch* coverage — which, as defined in the literature, means the execution of every branch of the program (limited here to reachable branches). This notion of branch coverage does not do full justice to loops, one of the most important and delicate programming constructs, since it only requires the test suite to include, for each loop `until e loop b end`, one test that executes the loop body b (with the exit condition e not satisfied) and one test that does not (with e satisfied). In other words, it treats the loop, for testing purposes, in the same way as a conditional `if not e then b else fail end`. (In this section we use `fail` as an abbreviation for `check False end`, i.e. an instruction that always fails.)

A loop is of course more than a conditional; its body may in full generality be executed any number of times, not just zero or one. Many important bugs will remain undetected by tests that do not execute the body more than once. To correct this limitation, we extend the SC strategy with a *loop unrolling* strategy, called “SCU” (Seeding Contradiction + Unrolling).

2.3.1 Basic SCU idea

For a natural integer n , we define the **n -unrolling** of a loop as a restricted version of that loop that allows up to n executions of the body. The conditional equivalent, used for branch coverage, is in this terminology a 1-unrolling, but in the general case n is a user-settable parameter. The idea of SCU is to replace loops by their n unrollings in the generated tests.

The choice of n is a tradeoff between the regression-bug-detection power of the generated tests, which increases (as the results in section 5 will show) with n , and testing time, which also grows with n .

Before examining the details of this tradeoff on the basis of experimental results, it is useful to define the concept of unrolling more precisely.

2.3.2 Theoretical basis for SCU

The concept of unrolling relies on a formal theory of loops. Since the details of the theoretical discussion are not essential for the present article, they have been relegated to a companion technical note, but here is the basic idea, without the full mathematical framework and the proofs. The approach considers a loop as the union (similar to a limit) of a sequence of programs defined as follows:

$$L_0 = \text{fail --i.e. check False end}$$

$$L_{i+1} = \text{if } \neg e \text{ then } B; L_i \text{ end } (i \geq 0)$$

In other words, L_i is of the form

```

if not e then
  B
  if not e then
    B
    if not e then
      ...
      if not e then
        B
        check False end
      end
    end
  end
end
end
end

```

L_i is what has been called an i -unrolling above.

Reference [20] defines the operations formally and proves that the above expansion does define a correct sequence of increasingly precise approximations of the loop.

3 Correctness

The goal of a test-suite-generation strategy is to produce high-coverage test suites. The Seeding Contradiction strategy is more ambitious: we consider it correct if it achieves **exhaustive coverage** (as defined in section 1: full coverage of reachable branches). More precisely, we will now prove that SC is “coverage-complete” if the prover is “reachability-sound”, “correctness-sound” and “counterexample-complete”. 3.1 defines these concepts and 3.2 has the proof.

3.1 Definitions and assumptions

Establishing the correctness of SC requires precise conventions and terminology.

A general assumption is the availability of an “FP-FT” mechanism which, as described in previous articles [9], can produce directly executable tests (expressed in the target programming language, in our case Eiffel) from counterexamples produced by the SMT-based prover. As a consequence, the rest of this discussion does not distinguish between the notions of counterexample and test.¹

The definition of basic block, or just **block** for short, appeared earlier (2.2).

For simplicity, we assume that the programs are **structured**, meaning that they use sequences, loops and conditionals as their only control structures. Also, we consider that a conditional always includes exactly one “else” part (possibly empty), and that a loop has two blocks, the loop body and an empty block (corresponding to the case of zero iterations). Further, expressions, particularly conditional expressions used in conditional instructions, are side-effect-free. Thanks to these conventions, instruction coverage (also known as statement coverage) and branch coverage are the same concept, called just “coverage” from now on.

A (possibly empty) block of a program is **reachable** if at least one set of input values will cause it to be executed, and otherwise (if, regardless of the input, it cannot be executed) *unreachable*. Reachability is an undecidable property for any realistic programming language, but that need not bother us since this work relies on a prover of which we will only require that it be **reachability-sound**: if a block is reachable, the prover will indeed characterize it as reachable. (The prover might, the other way, wrongly characterize a block as reachable when in fact it is not: with `if cos2(x) + sin2(x) = 100 then y := 0 else y := 1 end`, the prover might consider `y = 0` as a possible outcome if it does not have enough built-in knowledge about trigonometric functions. That too-conservative determination does not endanger the SC strategy.)

A program may contain instructions of the form `check C end`, with no effect on execution (as previewed in section 2). Such an instruction is **correct** if and only if the condition `C` will hold on every execution of the instruction. This property is again undecidable, and again we only need the prover to be **correctness-sound**: if it tells us that an instruction is correct, it is. (We hope the other way around too, but do not require it.) For the SC strategy we are interested in the trivial case for which `C` is `False`.

Also for simplicity, we assume that all correctness properties are expressed in the form of `check` instructions; in particular, we replace any contract elements (preconditions, postconditions, loop invariants and variants, class invariants) by such instructions added at the appropriate places in the program text.

With this convention, a **block** is correct if all its `check` instructions are, and a **program** is correct if all its blocks are. For a normally written program, this definition means that the program is correct in the usual sense; in particular, if it has any

¹Counterexamples that the prover generates at first can use arbitrary values, sometimes too large to be meaningful to programmers; as noted in 2.2, a minimization strategy is available to produce more intuitive values. The SC technique and its analysis are independent of such choices of counterexamples.

contracts, it satisfies them, for example by having every routine ensure its postcondition. The SC strategy, by adding `check False end` to individual blocks, makes these blocks — and hence the program as a whole — incorrect.

A **test suite** is a collection of test cases for a program.

A test suite achieves **exhaustive coverage** if for every reachable block in the program at least one of its test cases causes that block to be executed. (Note the importance of having a reachability-sound prover: if it could wrongly mark some reachable blocks as unreachable, it could wrongly report exhaustive coverage, which is not acceptable. On the other hand, if it is reachability-sound, it may pessimistically report less-than-exhaustive coverage for a test suite whose coverage is in fact exhaustive, a disappointing but not lethal result. This case does not occur in our examples thanks to the high quality of the prover.)

A test-suite-generation method (such as Seeding Contradiction) is **coverage-complete** if the generated test suite achieves exhaustive coverage for any correct program. In other words, for each reachable basic block of a correct program, at least one test in the suite will execute the block.

Finally, consider a prover that can generate counterexamples for programs it cannot prove correct. The prover is **counterexample-complete** if it generates a counterexample for every block that it determines to be reachable and incorrect.

With these conventions, the correctness of the Seeding Contradiction method is the property (proven next) that

If the prover is reachability-sound, correctness-sound and counterexample-complete, SC is coverage-complete.

3.2 Proof of correctness

To establish that correctness holds, on the basis of the preceding definitions, we first establish the following two lemmas:

- 1 Any test case of a seeded program (the program modified by addition of `check` instructions as described above) yields, by omitting the `bn` variable, a test case of the original program, exercising the same basic block.
- 2 Any reachable block of the original program is reachable in the seeded one.

The proof of both lemmas follows from the observation that the seeded program has the same variables as the original except for the addition of the `bn` variable, which only appears in the conditional `check` instructions and hence does not affect the behavior of the program other than by possibly causing execution of one of these instructions in the corresponding block. If `bn` has value `i` in such an execution, the execution of all blocks other than the block numbered `i` (if any — remember that we accept the value 0 for `bn`), in particular the execution of any block in an execution path *preceding* the possible execution of block `i`, proceeds exactly as in the original unseeded program. As a result:

- Any test executing block number `i` in the seeded program for any `i` has, for all other variables (those of the original program), values that cause execution of block `i` in the original program too, yielding Lemma 1.
- Consider a reachable block, numbered `i`, of the original program. Since it is reachable, there exists a variable assignment, for the variables of the original program,

that causes its execution. That variable assignment complemented by `bn = i` causes execution of block `i` in the seeded program, which is therefore reachable, yielding Lemma 2.

To prove that SC satisfies the definition of correctness (given at the end of 3.1):

- Assume that the original program is correct; then the only incorrect instructions in the seeded program are the added conditional `check` instructions (the `if C then check False end` at the beginning of every block).
- Consider an arbitrary reachable basic block `B`, of the original program. Because of Lemma 2, it is also reachable in the seeded program.
- If the prover is reachability-sound, it indeed determines that block `B` is (in the seeded program) reachable.
- If the prover is also correctness-sound, it determines that `B`'s seeded `check` instruction is incorrect, and hence (by definition) that `B` itself is incorrect.
- Then if it is counter-example-complete it will generate a counterexample that executes `B` in the seeded program.
- By Lemma 1, that counterexample yields a test that executes block `B` in the original program.
- As a consequence, by the definition of correctness above, the Seeding Contradiction strategy is correct.

3.3 Correctness in practice

To determine that SC as implemented is correct, we depend on properties of the prover: the definition assumes that the prover is reachability-sound, correctness-sound and counterexample-complete.

To our knowledge, no formal specification exists for the relevant tools in our actual tool stack (Fig. 1), particularly Z3 and Boogie. In their actual behavior as observed pragmatically, however, the tools satisfy the required properties.

4 Implementation

We have implemented Seeding Contradiction strategy in the form of a new option of the AutoProof program-proving framework, called “Full-coverage Test Generation” (FTG)². The implementation relies on the FP-FT [9] [10] feature of AutoProof, which enables automatic generation of failed tests from failed proofs. The objective is to add the incorrect `check` instructions at the appropriate program locations so that the verification of the seeded program results in proof failures, yielding an exhaustive-coverage test suite as described above.

Like the rest of AutoProof, seeding is modular: routine by routine. It is applied at the Boogie level, so that the Eiffel program remains untouched. The Boogie equivalent of the `check` instruction is written `assert`. Depending on the structure of the code for a routine `r`, five cases arise, reviewed now.

²AutoProof including the FTG option is available for download at github.com/huangl223/ES-AP-Installation.

A - Plain Block. If the body of r includes no conditional and hence has only one path, the SC strategy inserts a single `assert false` at the beginning of the body. Verification of r results in failure of the assertion; by applying FP-FT, we obtain a valid test case of r (whose test input satisfies the precondition).

B - Implicit else branch. If r contains a conditional whose `else` branch is implicit, SC makes it explicit and produces a test case covering the branch. Fig. 2 shows an example: SC inserts two `assert` clauses, one in the `then` branch and the other in the `else` branch that it creates. Running the proof produces two counterexamples for the two injected `assert` clauses, hence two tests.

r	<pre> do B0 if c then B1 end B2 end </pre>	<pre> implementation r () { T(B0) if (c){ assert false; T(B1) } else{ assert false; } T(B2) } </pre>
-----	--	--

Fig. 2: Instrumentation for r with implicit `else` branch. Left: original Eiffel code of r . Right, seeded Boogie code. B_i ($i \in \{0, 1, 2\}$) is a basic block in Eiffel, c a branch predicate evaluating to `true` or `false`, $T(B_i)$ the Boogie translation of B_i .

C - Cascading Branches. If r has a series of branches placed sequentially, as in Fig. 3, the SC algorithm inserts an `assert false` clause in each branch. The resulting tests cover all branches.

r	<pre> do B0 if c1 then B1 elseif c2 then B2 else B3 end B4 end </pre>	<pre> implementation r (){ T(B0) if (c1){ assert false; T(B1) } else{ if (c2){ assert false; T(B2) } else{ assert false; T(B3) } } T(B4) } </pre>
-----	---	---

Fig. 3: Instrumentation for cascading branches: three `assert false` clauses are inserted for the three branches in r ; note that the `elseif` instruction in Eiffel, together with the last `else` instruction, is mapped to a nested `if-else` instruction in Boogie.

D - Nested branches. When conditionals are nested, SC only generates tests targeting the *leaf* branches — those with no embedded conditionals. This approach is sound since any program execution that exercises a leaf branch must also go through all the branches leading to it. Fig. 4 has three leaf branches for blocks $B2$, $B3$ and

B5. Any execution going through B2 and B3 will exercise B1; SC only inserts `assert` instructions for leaves (none for B1).

```

r do
  B0
  if c1 then
    B1
    if c2 then
      B2
    else B3 end
    B4
  else B5 end
  B6
end

implementation r (){
  T(B0)
  if (c1){
    T(B1)
    if (c2){
      assert false; T(B2) }
    else{
      assert false; T(B3) }
    T(B4) }
  else{
    assert false; T(B5) }
  T(B6) }
}

```

Fig. 4: Instrumentation for nested branches

E - Sequential decisions. If `r` has multiple successive decision instructions, as in Fig. 5, SC inserts the conditional `assert false` instructions as explained in 2.2. It declares a variable `bn` for the block number and adds “`if (bn == i) assert false;`”. Since the value of `bn` is between 0 and `N` (number of target blocks), it adds a clause “`requires bn ≥ 0 && bn ≤ N`” to the precondition of `r`.

```

r do
  B0
  if c1 then
    B1
  else B2 end
  B3
  if c2 then
    B4
  else B5 end
  B6
end

implementation r (){
  int bn;
  T(B0)
  if (c1){
    if (bn == 1) assert false; T(B1) }
  else{
    if (bn == 2) assert false; T(B2) }
  T(B3)
  if (c2){
    if (bn == 3) assert false; T(B4) }
  else{
    if (bn == 4) assert false; T(B5) }
  T(B6) }
}

```

Fig. 5: Instrumentation for sequential conditionals

5 Evaluation and comparison with other test generation techniques

We evaluate the performance of Seeding Contradiction as implemented in AutoProof per the preceding section, comparing it to other existing test generation tools: IntelliTest [3] (previously known as Pex, a symbolic execution test-generation tool for .NET), a model checking tool CPAChecker [21, 22], and AutoTest [4], a test generation tool for Eiffel using Adaptive Random Testing, specifically ARTOO [23]). We also

evaluate the implementation of SCU and discuss the trade-offs between performance and unrolling depth as mentioned in section 2.3.1. In line with these two objectives, we formulate the following research questions:

RQ1 *How does the branch coverage achieved by SC and the other tools?*

RQ2 *What is the average cost, in terms of time cost and test suite’s size, to achieve full branch coverage for SC and the other tools?*

RQ3 *Does loop unrolling actually lead to test suites that find more bugs?*

RQ4 *What is the precise effect of loop unrolling on verification time?*

5.1 Experiment design

To answer RQ1 and RQ2, the experiment compares performance of SC to three other test generation tools. The experiment applies all four tools to generate tests for 23 programs adapted from examples in the AutoProof tutorial³ and benchmarks of previous software verification competitions [24] [25] [26]. Table 1 lists their characteristics, including implementation size (number of Lines Of Code) and number of branches.

The examples are originally in Eiffel; we translated them manually into C# and C for IntelliTest and CPAChecker respectively. The experiment includes a test generation session for every example in every tool. For AutoTest, whose algorithms keeps generating tests until a preset time limit, it uses 10 minutes (600 seconds) as that limit; there is no time limit for the other two approaches.

To answer RQ3 and RQ4, the experiment uses a set of 5 classes that contain loops and applies SC to generate tests for those classes, with loops unrolled to different depths. It then compares the fault-identification performance of the resulting test suites. For each class, the experiment consists of 12 different groups, with the unrolling depth i set to 0, 1, 2, ..., 10, 15, respectively. For each of the 5 classes, the experiment creates 5 different faulty variants by randomly injecting errors into the correctly verified version, which totalled 25 variants. To assess the overall performance of each unrolling group, it then performs for each group 10 repetition runs of the following procedures: generate tests; run the tests on the faulty variants; collect the faults found during testing.

All sessions took place on a machine with a 2.1 GHz Intel 12-Core processor and 32 GB of memory, running Windows 11 and Microsoft .NET 7.0.203. Versions used are: EiffelStudio 22.05 (used through AutoProof and AutoTest); Boogie 2.11.10; Z3 solver 4.8.14; Visual Studio 2022 (integrated with IntelliTest); CPAChecker 3.0. All code and results are available at <https://github.com/huangl223/Seeding-contradiction>.

5.2 Analyses and results

Branch coverage (RQ1) To answer our RQ1: *How does the branch coverage achieved by SC and the other tools?*, we measure the branch coverage achieved by the four tools. For each example, we executed the generated test suite and calculated coverage as the ratio of *number of exercised branches* over *number of branches*. Table 2 shows coverage results.

³<http://autoproof.sit.org/autoproof/tutorial>

Table 1: Examples

Example	Lines of Code	Number of Branches
Account	214	14
Clock	153	8
Heater	102	8
Lamp	95	8
Max	49	3
Linear search	64	5
Insertion sort	122	5
Gnome Sort	62	5
Square root	56	5
Sum and max	56	4
Arithmetic	204	14
Binary search	74	5
Recursive binary search	89	7
Dutch flag	188	11
Two way max	49	4
Two way sort	85	6
Quick sort	232	9
Selection sort	167	5
Bubble sort	165	5
Optimized gnome sort	183	8
Queue	239	16
Stack	227	15
Prime numbers	129	6
Total	3004	176

SC and IntelliTest handle the examples effectively, achieving coverage near 100%. SC achieves exhaustive coverage (100% of reachable branches) for all 23 examples, IntelliTest achieves it for 22 examples, and CPAChecker achieves it for 18 examples. AutoTest, due to its random core, achieves the lowest coverage, reaching exhaustive coverage for only 7 examples. The maximum possible for `Lamp` is 87.5% as it contains an unreachable branch. IntelliTest reaches exhaustive coverage for most examples but misses it for `Account` and `Lamp`. CPAChecker cannot achieve 100% coverage the examples that involve recursion (`Arithmetic`, `Recursive binary search`, `Quick sort`): it reports recursion as an unsupported feature and aborts the test generation process. AutoTest’s coverage varies from 50% to 100%. Occasionally, it performs better than IntelliTest, reaching the maximum 87.5% for `Lamp` against IntelliTest’s 50%.

Performance (RQ2) To answer RQ2: *What is the average cost, in terms of time cost and test suite’s size, to achieve full branch coverage for SC and the other tools?*, we measure the time cost and test suite’ size for each tool. Table 3 and Table 4 present the results.

Table 2: Result: branch coverage

Example	SC	IntelliTest	CPAChecker	AutoTest
Account	100%	92.85%	100%	78.6%
Clock	100%	100%	100%	70%
Heater	100%	100%	100%	62.5%
Lamp	87.5%	50%	87.5%	87.5 %
Max	100%	100%	100%	66.7%
Linear search	100%	100%	100%	100%
Insertion sort	100%	100%	100%	80%
Gnome sort	100%	100%	100%	60%
Square root	100%	100%	87.5%	100%
Sum and max	100%	100%	50%	100%
Arithmetic	100%	100%	65.38%	100%
Binary search	100%	100%	100%	100%
Recursive binary search	100%	100%	28.57%	85.7%
Dutch flag	100%	100%	100%	72.7%
Two way max	100%	100%	100%	75%
Two way sort	100%	100%	100%	83.3%
Quick Sort	100%	100%	75%	100%
Selection sort	100%	100%	100%	100%
Bubble sort	100%	100%	100%	80%
Optimized gnome sort	100%	100%	100%	50%
Queue	100%	100%	100%	90.91%
Stack	100%	100%	100%	90%
Prime numbers	100%	100%	100%	66.7%
Average	99.4%	97.5%	91%	81.2%

Table 3 gives the time needed to produce the test suite in the various approaches, using the following conventions:

- For SC, time for test generation includes two parts: proof time (for AutoProof) and time for extracting tests from failed proofs (time for FP-FT).
- For AutoTest, the time is always the 10-minute timeout, chosen from experience: within that time, test generation of examples usually reaches a plateau.
- CPAChecker does not offer directly the duration for test generation. Instead, we measure this duration by subtracting the time spent on analysis, such as bounded model checking, from the total elapsed time.
- IntelliTest does not directly provide time information. We measure duration manually by recording the the timestamps of session start and termination.

As the results in Table 3 show, static approaches (SC and CPAChecker) outperform dynamic approaches (IntelliTest and AutoTest). Among these tools, SC achieves exhaustive coverage significantly faster than the others, requiring approximately 0.5 seconds on average. This is about half the time needed by CPAChecker, 50 times faster than IntelliTest, and 500 times faster than AutoTest.

Table 3: Result: time (in seconds) to reach exhaustive coverage

Example	SC	IntelliTest	CPAChecker	AutoTest
Account	0.56	9.58	3.88	–
Clock	0.44	7.44	1.03	–
Heater	0.85	8.06	0.98	–
Lamp	0.39	–	0.98	233.03
Max	0.37	8.19	1.01	–
Linear search	0.42	11.77	0.99	–
Gnome sort	0.52	10.89	1.16	–
Square root	0.26	12.86	–	1.71
Sum and max	0.37	10.99	–	1322.61
Arithmetic	0.42	32.98	–	14.49
Binary search	0.44	99.29	1.00	150.86
Recursive binary search	0.48	13.07	–	–
Dutch flag	0.43	31.36	1.06	330.89
Two way max	0.52	9.59	0.97	–
Two way sort	0.39	80.91	1.04	–
Quick sort	0.90	111.57	–	78.37
Selection sort	0.50	17.81	1.03	–
Bubble sort	0.59	14.74	1.00	–
Optimized gnome sort	0.54	12.32	1.01	–
Queue	0.89	9.87	1.14	–
Stack	0.41	28.39	1.08	–
Prime numbers	2.06	5.91	1.10	–
Average	0.554	24.9	1.14	304.57

SC is the fastest among the four tools, with all its test generation runs taking less than 1 second. It generally takes longer for test generation when examples involve complex contracts, such as `Prime numbers`. The test generation for CPAChecker remains relatively stable, typically it takes about 1 second to complete. It takes longer for the `Account` example, which has the greatest number of dependencies between different functions. For IntelliTest, test generation takes less than 40 seconds for most examples, but three of the 23 examples (i.e. `Binary search`, `Two way sort`, `Quick sort`) require more than one minute. For AutoTest, test generation time varies from 1.71 seconds for `Square root` to more than 20 minutes for `Sum and max`.

The importance of minimizing the size of test suites has become a crucial concern [27]. From a software engineering viewpoint, particularly for the long-term health of a project, a smaller size achieving the same coverage is better, since it results in a more manageable test suite giving the project the same benefits as a larger one. Table 4 displays the number of generated tests to reach exhaustive coverage.

Table 4: Result: number of generated tests to reach exhaustive coverage

Example	SC	IntelliTest	CPAChecker	AutoTest
Account	13	13	13	–
Clock	10	13	10	–
Heater	8	8	10	–
Lamp	7	–	11	656
Max	3	4	4	–
Linear search	3	7	8	127
Insertion Sort	3	5	8	–
Gnome Sort	3	7	6	–
Square root	4	5	–	18
Sum and max	3	5	–	1784
Arithmetic	14	25	–	531
Binary search	4	6	12	905
Recursive binary search	7	15	–	–
Dutch flag	9	27	20	–
Two way max	2	4	4	–
Two way sort	5	9	6	–
Quick sort	9	18	–	342
Selection Sort	5	12	12	–
Bubble Sort	4	8	6	–
Optimized gnome sort	7	8	16	–
Queue	19	46	30	–
Stack	16	24	26	–
Prime numbers	7	10	13	–
Average	7.17	12.68	11.9	623.28

Overall, SC generates the smallest test suites, with an average size of approximately 7 for exhaustive coverage. In comparison, the average size of the exhaustive-coverage test suites for IntelliTest and CPAChecker is similar, both averaging around 12. While these sizes are slightly larger than SC’s, they are substantially smaller than the test suites produced by AutoTest.

In most cases, the number of tests generated by SC is the same as the number of blocks: as each generated test results from a proof failure of an incorrect instruction, seeded at one program location, each test covers just the corresponding block and introduces no redundancy. If nested branches are present, the size of the test suite can actually be less than the number of branches: SC only generates tests targeting the innermost branches (the leaf nodes of the control structure), as explained in section 4; each test going through these branches automatically covers all its enclosing branches.

IntelliTest also generates small test suites, but is slower. The reason is IntelliTest’s use of concolic testing [28], which tests all feasible execution paths: since a branch can occur in several paths, a test will often identify a branch that was already covered by a different path. AutoTest, for its part, produces much larger test suites: as an Adaptive Random Testing tool, it often generates multiple test cases covering the same branches.

The above experimental results for RQ1 and RQ2 provide evidence of the benefits of the approach (subject to the limitations examined in the next section): SC is fast and efficient; it uses less than 1 second to produce an exhaustive-coverage test suite with the fewest number of test cases. Other observations include the following:

- AutoTest does not guarantee that the test inputs satisfy the routine’s precondition, while SC and IntelliTest always generate precondition-satisfying test inputs. The reason is that SC and IntelliTest rely on the results of constraint solving, where the routine’s precondition is encoded as an assumption and will always be satisfied.
- The SC approach has a prerequisite: the program under test has to be proved correct (the proof of the original program has no failure), while AutoTest and IntelliTest have no such constraint.
- As to the values of the generated test inputs, IntelliTest and AutoTest always apply small values that are easy to understand. SC initially produces test inputs that may contain large values; its “minimization” mechanism [10] corrects the problem.

Effectiveness of SCU (RQ3) To answer our RQ3: *Does loop unrolling actually lead to test suites that find more bugs?*, we run test suites generated with different loop unrolling depth and collect number of detected faults. A test suite generated in a run may contain several test cases that uncover the same failure (violation of the same contract) multiple times. To avoid the resulting redundancy, the experiment only collects *distinct* faults. A distinct fault is identified by a unique tuple:

< program variant, tag of failed contract, line number of the failure >

The evaluation of the performance of fault detection in each unrolling group of a class c uses the following two criteria:

- N_c : the number of all distinct faults that appear during the 10 repetition runs of that group, which can be described as a function over the unrolling depth i :

$$N_c(i) = \left| \bigcup_{j=1}^{10} F_c(i, j) \right|$$

where $F_c(i, j)$ is the set of distinct faults found at the j^{th} run ($1 \leq j \leq 10$) with unrolling depth i for the class c . Informally, $N_c(i)$ represents the number of faults that can be found by unrolling group i when the experiment repeats the test generation a sufficient number of times.

- R_c : the number of distinct faults detected per run, which can be defined as a function:

$$R_c(i) = \frac{\sum_{j=1}^{10} |F_c(i, j)|}{10}$$

where $R_c(i)$ is the average performance of fault detection of unrolling group i . $R_c(i)$ is necessary for the assessment, as different test generation runs use different random seeds (for SMT solving), resulting in different test suites and hence in different detected faults.

Table 5 displays the experimental results: the first 5 columns list the value of N_c (left subcolumn) and R_c (right subcolumn) for the 5 classes; **Original** reports the number of faults detected using the original SC strategy (without SCU’s loop unrolling); **Unroll i** lists the number of faults found with the loop unrolled i times; **Total** sums up the number of faults of all examples. The last column reports the average time for test generation over all classes for each unrolling group.

Test suites produced without unrolling find 46 (83.6%) out of 55 distinct faults; the average number of detected faults per run is about 57.5% (28.5 out of 49.6) of the best performance. In other words, even without unrolling the generated tests are already good enough to uncover over 80% of all faults, although each test generation run detects a relatively small number of faults.

On the other hand, unrolling loops can indeed increase the number of faults found: unrolling loops 3 times leads to an improvement of N_c by 9.1% and R_c by 27.8%, with an acceptable overhead of 0.37 seconds on average; an unroll depth of 8 is sufficient to find 98% of all the distinct faults; unrolling 10 times obtains the maximum value for N_c and yields an improvement of R_c by 41.5%; unrolling 15 times achieves the best average performance (the value of R_c is the largest), although with a much larger overhead.

Fig. 6 depicts the changes of performance in percentage when the unroll depth increases; This percentage is computed by dividing N_c by the maximum number of faults found during all testing sessions of class c . The dashed line plots the mean of percentages over all examples. In all, when the unroll depth increases from 1 to 3, the percentage of faults found increases significantly; for **Two way max**, **Square root** and **Linear search**, increasing the depth further will still improve the percentage, while in the cases of **Binary search** and **Sum and max**, moderate degradation occurs at depth 4 and 7, respectively.

The circular dots represent the percentages of the **Original** group of the corresponding classes. The locations of the dots reflect the approximate unrolling depth needed to obtain the same level of performance. For example, the circular dots of **Linear search** and **Square root** locate at where x-coordinates close to 1; this indicates that, for those two classes, performance without unrolling are similar to that of unrolling once ($i = 1$). In the case of **Binary search**, the circular dot appears at the summit of the curve, meaning that the tests generated without unrolling seems good enough to detect the maximum number of faults. The example that benefits most from unrolling is **Linear search**, since it has both the shortest horizontal distance and the largest vertical distance between the circular dot and the summit.

Table 5: Performance of fault detection and verification time

	Sum and max		Two way max		Binary search		Linear search		Square root		Total		Avg. Time (sec)
Original	11	4.6	8	6	12	7.5	4	4	11	6.4	46	28.5	0.16
Unroll 1	5	4.6	3	2.4	9	6.1	4	4	11	7.8	32	24.9	0.18
Unroll 2	9	6.7	7	5.7	12	6.8	5	5	11	10.2	44	42.3	0.31
Unroll 3	11	8.9	10	7.9	12	9.9	7	6.7	11	8.9	51	42.3	0.53
Unroll 4	12	9	10	8.3	11	9.5	7	6.9	11	10.7	51	44.4	0.78
Unroll 5	12	9.7	10	8.7	12	10.2	7	7	11	9.7	52	45.3	1.25
Unroll 6	13	9.4	10	8.8	12	10.9	7	6.9	12	11.1	54	47.1	2.13
Unroll 7	12	9.3	10	9.1	12	10.9	7	7	12	11.2	53	47.2	3.58
Unroll 8	13	9.2	10	8.5	12	11.2	7	7	12	11.2	54	47.1	6.46
Unroll 9	13	8.8	10	9.1	12	11.8	7	7	12	9.5	54	46.2	21.85
Unroll 10	13	10.3	10	8.8	12	11	7	7	13	12	55	49.1	38.86
Unroll 15	13	10.1	10	9.6	12	11.6	7	7	13	11.3	55	49.6	301.57

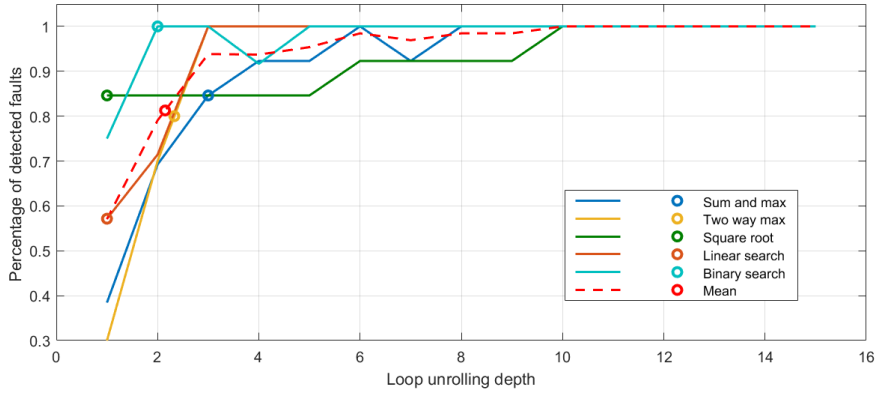


Fig. 6: Effect of loop unrolling on bug-finding performance

Fig. 7 shows the percentage of faults detected per run over all unrolling depths. The percentage is the ratio of R_c to the maximum number of faults detected for class c in a single run. The dashed curve represents the mean for all examples. We see that loop unrolling is very profitable for depths up to 3, with each depth increment in that range significantly improving bug-detection performance; cumulatively, the performance for a depth of 3 is 27% higher than with non-unrolled SC. Beyond depth 3, the performance still increases but less dramatically, and it deteriorates slightly for a depth of 9.

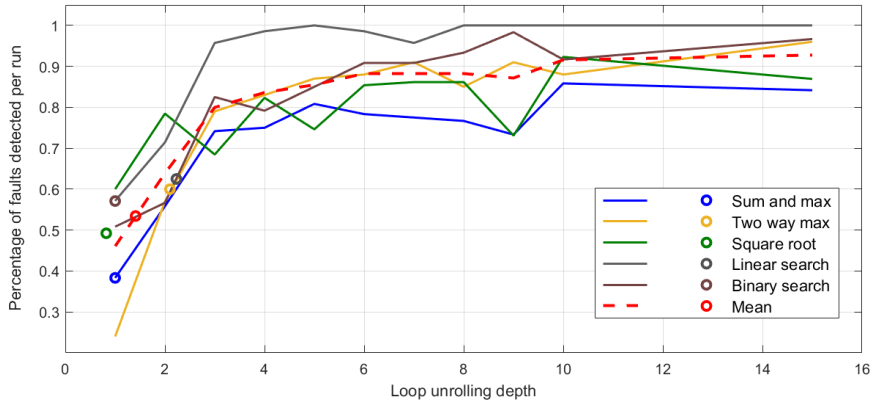


Fig. 7: Effect of loop unrolling on bug-finding performance, per test generation run

SCU Performance (RQ4) To answer RQ4: *What is the precise effect of loop unrolling on verification time?*, we measure the time of test generation with different unrolling depths. Fig. 8 plots the average time using a logarithmic scale. The overall overhead of unrolling (depicted by the dashed line) roughly grows exponentially. When the depth is below 5, however, it seems still acceptable: a test generation task costs only a few seconds for most of the classes. The overhead grows quickly when the depth ranges between 7 to 9, mostly affected by the growth of the **Binary search** class. Among the 5 classes, **Linear search** has the smallest growth, and its overhead is still within 1 second when the unrolling depth reaches 15. The reason is that the loop bodies of the other classes contain at least 2 branches, while **Linear search** contains only one instruction and has no multiple branches; as a consequence, when SC generates tests for **Linear search**, fewer contradiction contracts would be added.

The above results, while still applied to a small set of examples, speak largely for themselves. It is striking to see how much branch coverage — the gold standard of the testing industry, which in fact often defines a goal of only 80%, with many teams satisfying themselves with much less — misses many bugs that unrolling discovers, and continues to discover as the unrolling depth increases.

For an automatic test generation such as SC, the unrolling extension seems an essential facility to add. Further research and validation will be welcome to explore this assertion further.

6 Limitations and threats to validity

The setup of the SC approach assumes a Hoare-style verification framework (of which Boogie is but one example), and the availability of a test generation mechanism that supports generating test cases from proof failures. We have not studied the possible application of the ideas to different verification frameworks, based for example on abstract interpretation or model checking.

The current version of SC is subject to the following limitations:

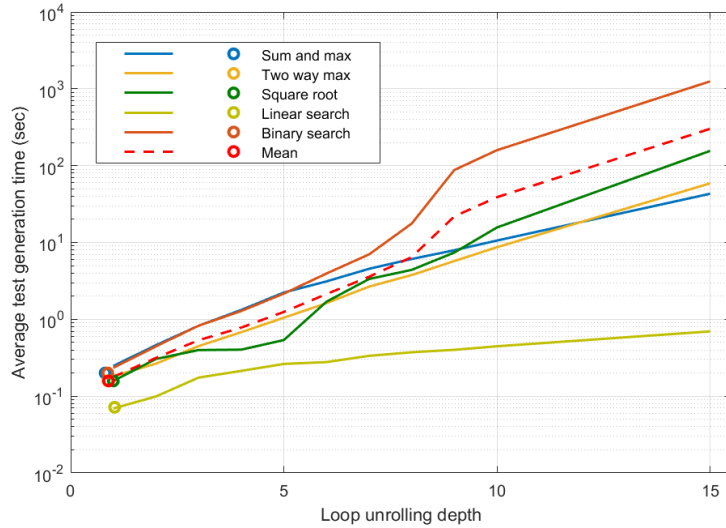


Fig. 8: Time taken on average (s) depending on the loop unrolling depth.

- SC is not able to handle programs with non-linear computations (such as derivation and exponentiation); this restriction comes from the underlying SMT solver.
- SC does not support the more advanced parts of the Eiffel system, in particular generic classes. Data structures are limited to arrays and sequences.
- The current implementation of SCU uses a user-specified unrolling depth, which is not sufficient to guarantee exhaustive coverage — if the unrolling depth is too small, then some branches may not be covered.

These limitations will need to be overcome for SC to be applicable to industrial-grade programs.

The following considerations may influence the generalization of the results achieved so far:

- The number of repeated experiments increased the potential threats to internal validity. We hope that further experiments with large number of iterations will provide more conclusive evidence.
- Although a few of the examples classes that we processed so far are complex and sophisticated, most are of a small size and not necessarily representative of industrial-grade object-oriented programs. In the future, we intend to use the Eiffel-Base library⁴, which has yielded extensive, representative results in the evaluation of AutoProof and AutoTest, and exhibits considerable variety and complexity in terms of size (according to various metrics), richness of program semantics, and sophistication of algorithms and software architecture.

⁴EiffelBase Data Structures: https://www.eiffel.org/doc/solutions/EiffelBase_Data_Structures_Overview

7 Related work

The present work is an extended version of [1], adding coverage of new examples, inclusion of loop unrolling, and numerous other additions and improvements.

Previous work has taken advantage of counterexamples generated by failing proofs, but for other purposes, in particular automatic program repair [29] and generation of failing tests [8] [9]. These techniques work on the original program and not, as here, on a transformed program in which *incorrect* instructions have been inserted with the express purpose of making the proof fail.

The earliest work we know to have applied this idea [30] [31] generates tests for low-level C programs using Bounded Model Checking (BMC) [32], producing test suites with exhaustive branch coverage. A more recent variant, for Java bytecode, is JBMC [33]. In contrast with SC, each verification run only activates one assertion at a time, producing one counterexample. This approach is conceptually similar, in the terminology of the present work (2.2), to the “MSP” (Multiple Seeded Programs) technique, although the C version [30] uses compile-time macros, one for each block, to avoid the actual generation of multiple programs. In contrast, the present work uses RSSP (Repeatedly Seeded Single Program), relying on a single *run-time* variable representing the block number. Moreover, SC performs one single verification run, which results in a collection of tests from all counterexamples. In contrast, the approach in [30] [31] also produces one file with multiple assertions added, but it requires execution of the model checker multiple times: for each run an assertion is activated at a time, thus each run produces a single test. BMC-based approaches rely on the correctness of the *bound* of the execution trace: if the bound is not set correctly, some branches might not be covered, requiring more verification runs to obtain a better bound.

Other techniques that apply constraint solving for generating inputs includes test generations based on symbolic execution, such as Pex/IntelliTest [3], KLEE [34], PathCrawler[35]. None of the strategies proposed guarantees exhaustive branch coverage; they can achieve it when a systematic test generation strategy, rather than one based on heuristics or randomization, is applied.

A very recent development (published just as the present work was being submitted) is DTest, a toolkit [36] for generating unit tests for Dafny programs, applying ideas similar to those of SC. As the generated Dafny tests are not directly executable, test generation requires transformation of Dafny programs and tests into a mainstream language. In contrast, the present approach works directly on Eiffel programs. The DTest coverage results cited in the referenced article are 100% on only 2 of its examples, and go down to as low as 58% on the others. One should not draw definite conclusions from these figures, since the examples are different, their program sizes too (more precisely, most of the examples are of comparable sizes, but the cited work has three between 1100 and 1900 LOCs, which we have not handled yet), and the article does not mention any presence of unreachable code (which makes it impossible to distinguish between full coverage and exhaustive coverage). It should be noted, however, that the article also makes no mention of the “Seeded Unreachability” issue discussed in section 2.2; in fact, it states that “*DTest enters a loop where it systematically injects trivially failing trap assertions (meaning assert false)*”, a technique which generally leads, for any program with a non-trivial control structure, to Seeded

Unreachability and hence to decreased coverage. That omission may be the reason for the relatively low coverage results reported in the article. The Conditional Seeding technique of SC, introduced by the present work, addresses Seeded Unreachability and has made it possible to reach exhaustive coverage in all examples so far. In addition, to obtain small test suites, DTest seems to require a separate minimization strategy, which takes from 8 to 1860 seconds on the cited examples, far beyond the times of running SC. In discussing minimization, the authors appear to come close to recognizing the Seeded Unreachability issue, without using the Conditional Seeding technique, when they write that “*we determine the feasibility of a path via a query to the SMT solver, in which a trap assertion is added that fails only if all the blocks along the path are visited*”, a technique that is “*exponential in the number of SMT queries (running on all benchmarks [cited in the article] would take weeks)*”. SC does not appear to need any such technique.

8 Conclusions and future work

This article presents Seeding Contradiction (SC), a tool that automatically generates test suites that achieve exhaustive branch coverage very fast. The presentation of the approach comes with a proof of correctness, defined as the guarantee that the generated test suite achieves exhaustive coverage (full coverage of reachable branches). While technical limitations remain, the evaluation so far demonstrates the effectiveness and efficiency of the SC approach through the comparison with two existing test generators IntelliTest and AutoTest, in terms of achieved coverage, generation time, and size of the test suite.

Ongoing work includes handling larger examples, processing entire classes instead of single routines, providing a mechanism to generate tests covering branches that a given test suite fails to cover, and taking advantage of the SC strategy to identify dead code. Exploring the minimum unrolling depth to guarantee full coverage is also under investigation.

Acknowledgements. We are particularly grateful, for their extensive and patient help, to Yi Wei (AutoTest) and Jocelyn Fiat (EiffelStudio and AutoProof). The paper benefitted from perceptive comments by the anonymous referees on the original version.

Conflict of interest. On behalf of all authors, the corresponding author declares that there is no conflict of interest.

References

- [1] Huang, L., Meyer, B., Oriol, M.: Seeding contradiction: a fast method for generating full-coverage test suites. In: IFIP International Conference on Testing Software and Systems, pp. 52–70 (2023). Springer
- [2] Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for java. In: Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, pp. 815–816 (2007)

- [3] Tillmann, N., De Halleux, J.: Pex–White Box Test Generation for .Net. In: International Conference on Tests and Proofs (TAP), pp. 134–153 (2008). Springer
- [4] AutoTest. https://www.eiffel.org/doc/eiffelstudio/Using_AutoTest
- [5] Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on java predicates. ACM SIGSOFT Software Engineering Notes **27**(4), 123–133 (2002)
- [6] Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., *et al.*: The key platform for verification and analysis of java programs. In: Verified Software: Theories, Tools and Experiments: 6th International Conference, VSTTE 2014, Vienna, Austria, July 17–18, 2014, Revised Selected Papers 6, pp. 55–71 (2014). Springer
- [7] Sen, K., Agha, G.: Cute and jcute: Concolic unit testing and explicit path model-checking tools: (tool paper). In: Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17–20, 2006. Proceedings 18, pp. 419–423 (2006). Springer
- [8] Nilizadeh, A., Calvo, M., Leavens, G.T., Cok, D.R.: Generating Counterexamples in the Form of Unit Tests from Hoare-style Verification Attempts. In: International Conference on Formal Methods in Software Engineering (FormaliSE), pp. 124–128 (2022). IEEE
- [9] Huang, L., Meyer, B.: A Failed Proof Can Yield a Useful Test. arXiv:2208.09873 (2022)
- [10] Huang, L., Meyer, B., Oriol, M.: Improving Counterexample Quality from Failed Program Verification. In: International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 268–273 (2022). IEEE
- [11] Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-active Functional Verification of Object-Oriented Programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 566–580 (2015). Springer
- [12] AutoProof. <http://autoproof.sit.org/>
- [13] Meyer, B.: Object-Oriented Software Construction, second edition. Prentice Hall (1997)
- [14] Leino, K.R.M., Rümmer, P.: The Boogie 2 Type System: Design and Verification Condition Generation. <https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.146.4277>
- [15] Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A

- Modular Reusable Verifier for Object-Oriented Programs. In: Int. Symposium on Formal Methods for Components and Objects, pp. 364–387 (2005). Springer
- [16] De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 337–340 (2008). Springer
- [17] Leino, K.R.M.: Program Proofs. MIT Press (2023)
- [18] Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. ACM SIGSOFT Software Engineering Notes **24**(6), 146–162 (1999)
- [19] Rayadurgam, S., Heimdahl, M.P.E.: Coverage based test-case generation using model checkers. In: Proceedings. Eighth Annual IEEE International Conference and Workshop On the Engineering of Computer-Based Systems-ECBS 2001, pp. 83–91 (2001). IEEE
- [20] Meyer, B.: Loop unrolling (for test coverage): formal definition. arXiv (2024)
- [21] Jakobs, M.-C.: Coveritest: interleaving value and predicate analysis for test-case generation. International journal on software tools for technology transfer **23**, 847–851 (2021)
- [22] Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14–20, 2011. Proceedings 23, pp. 184–190 (2011). Springer
- [23] Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: ARTOO: Adaptive Random Testing for Object-Oriented Software. In: International Conference on Software Engineering (ICSE), pp. 71–80 (2008)
- [24] Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., Frazier, D.: Incremental Benchmarks for Software Verification Tools and Techniques. In: Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE), pp. 84–98 (2008). Springer
- [25] Bormer, T., Brockschmidt, M., Distefano, D., *et al.*: The COST IC0701 Verification Competition. In: International Conference on Formal Verification of Object-Oriented Software (FoVeOO), pp. 3–21 (2011). Springer
- [26] Klebanov, V., Müller, P., *et al.*: The 1st Verified Software Competition: Experience Report. In: International Symposium on Formal Methods (FM), pp. 154–168 (2011). Springer
- [27] Orso, A., Hsu, H.-Y.: MINTS: A General Framework and Tool for Supporting

- Test-suite Minimization. In: International Conference on Software Engineering (ICSE), pp. 419–429 (2009)
- [28] Sen, K., Marinov, D., Agha, G.: CUTE: a Concolic Unit Testing Engine for C. In: The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC-FSE), pp. 213–223 (2005)
- [29] Nilizadeh, A., Calvo, M., Leavens, G.T., Le, X.-B.D.: More Reliable Test Suites for Dynamic APR by Using Counterexamples. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 208–219 (2021). IEEE
- [30] Angeletti, D., Giunchiglia, E., Narizzano, M., Palma, G., Puddu, A., Sabina, S.: Improving the Automatic Test Generation Process for Coverage Analysis Using CBMC. In: International RCRA Workshop (2009)
- [31] Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Automatic Test Generation for Coverage Analysis Using CBMC. In: International Conference on Computer Aided Systems Theory (EUROCAST), pp. 287–294 (2009). Springer
- [32] Kroening, D., Tautschnig, M.: CBMC–C Bounded Model Checker: (Competition Contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 389–391 (2014). Springer
- [33] Brenguier, R., Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: A Bounded Model Checking Tool for Java Bytecode. arXiv:2302.02381 (2023)
- [34] Cadar, C., Dunbar, D., Engler, D.R., *et al.*: KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI), vol. 8, pp. 209–224 (2008)
- [35] Williams, N.: Towards Exhaustive Branch Coverage with PathCrawler. In: Int. Conference on Automation of Software Tests (AST), pp. 117–120 (2021). IEEE
- [36] Fedchin, A., Dean, T., Foster, J.S., Mercer, E., Rakamaric, Z., Reger, G., Rungta, N., Salkeld, R., Wagner, L., Waldrip, C.: A Toolkit for Automated Testing of Dafny (2023)