Loop unrolling: formal definition and application to testing

Li Huang[1][0000−0003−3531−4045], Bertrand Meyer[1][0000−0002−5985−7434], and Reto Weber[1][0009−0001−9262−4843]

Constructor Institute of Technology, Schaffhausen, Switzerland
https://institute.constructor.org
first.last@constructor.org

Abstract. Testing processes usually aim at high coverage, but loops severely limit coverage ambitions since the number of iterations is generally not predictable. Most testing teams address this issue by adopting the extreme solution of limiting themselves to *branch coverage*, which only considers loop executions that iterate the body either once or not at all. This approach misses any bug that only arises after two or more iterations.

To achieve more meaningful coverage, testing strategies may *unroll* loops, in the sense of using executions that iterate loops up to n times for some n greater than one, chosen pragmatically in consideration of the available computational power.

While loop unrolling is a standard part of compiler optimization techniques, its use in testing is far less common. Part of the reason is that the concept, while seemingly intuitive, lacks a generally accepted and precise specification. The present article provides a formal definition and a set of formal properties of unrolling. All the properties have mechanically been proved correct (through the Isabelle proof assistant).

Using this definition as the conceptual basis, we have applied an unrolling strategy to an existing automated testing framework and report the results: how many more bugs get detected once we unroll loops more than once?

These results provide a first assessment of whether unrolling should become a standard part of test generation and test coverage measurement.

Keywords: testing · loop-unrolling · test-coverage

1 Loops, coverage and unrolling

The issue addressed in this work, both theoretically and empirically, is a basic question of software testing: is branch coverage, the prime practical measure of testing effectiveness, justified in its drastic simplification of treating a loop like a conditional, whose body is executed either once or not at all, even though in an actual program run the body can be executed any number of times? Should we instead "unroll" loops, improving the approximation by considering not just zero or one but any number of iterations, up to a set limit? We will first define

this concept of unrolling rigorously through a mathematical model. Then, using a set of practical examples and an automated test-generation framework relying on formal verification, we will examine how much (if anything) testing strategies miss when they limit themselves to standard branch coverage and, conversely, how many more bugs we can find if we unroll loops.

1.1 Loops

A key property of computers is their ability to repeat operations, often many times. The corresponding construct in programming languages is the loop. (Functional languages use recursion or an equivalent mechanism instead, but this discussion assumes an imperative language.) In its general form, a typical loop L may be written `until e loop B end`, with the following execution behavior: evaluate `e` (a boolean expression); if its value is True, do nothing; if its value is False, execute `B` (the "loop body", an instruction or sequence of instructions) and repeat the entire process from the beginning. The loop can also be written `while c loop B end` where `c` is the logical negation of `e`. The two forms are equivalent; this discussion will stick to the `until` variant. It can be convenient to include an initialization clause with the keyword `from`.

A characteristic of such loop constructs is that it is impossible to predict statically (in other words, from the program text) how many iterations of `B` a particular execution of the loop will produce; different executions, with different input data, may result in different numbers of iterations. (If the program is buggy, the loop may also fail to terminate after a finite number of iterations.) This unpredictability is one of the key challenges of software verification, particularly automatic test generation and associated measures of test coverage.

1.2 Branch coverage

Defined broadly, test coverage is a criterion assessing what share of a program's potential executions a given test suite (a set of tests for the program) exercises. The reason for defining coverage measures is that if we want to use testing to estimate the quality of the code, and more specifically the number of **remaining** bugs (as opposed to using testing just for finding bugs [16]), we face the obvious obstacle that *any* realistic program has an infinite or intractably large number of possible executions, forcing us to select [19] a small subset of them — the test suite — for the test campaign; but we need to have some idea of how representative the test suite is of the full set. While many measures of test coverage have been proposed (see e.g. [1] for a survey), by far the most commonly used in industry is branch coverage, which measures the percentage of the program's possible *control paths* (paths in the control flow of the program) being exercised. "Achieving branch coverage" means reaching 100% of those possible paths; in practice, many development teams in industry set a lower percentage, such as 80%, as the condition for shipping a product. While empirical studies have uncovered the limits of branch coverage, showing in particular [24] (see also [21] and [6]) that an extensive testing campaign can reach a plateau at over

90% coverage then continue to find bugs long after that stage, they have not affected the status of branch coverage as a key criterion in practical software development.

The definition of branch coverage used in practice makes a critical simplification with respect to loops. The obvious purpose is to skirt the major issue mentioned above, the impossibility of predicting how many times a loop will be iterated. The simplification is, however, drastic: branch coverage considers only two paths for a loop (Fig. 1), one which executes `B` once, and one that exits immediately. In other words, it reduces the loop **until** `e` **loop** `B` **end** to a simple conditional instruction **if not** `e` **then** `B` **end**.

In the reality of program execution, the set of possible paths is infinite, following the upward arrow of Fig. 1 an arbitrary number of times $\mathtt{n} \geq \mathtt{0}$. As a proxy for actual executions, branch coverage misses cases in which `n` is 2 or more. Since loops are essential to computing, it is remarkable that such a brutal simplification has not prevented branch coverage from achieving in software development a role that industry massively finds essential. It is legitimate, however, to ask how much we may be losing by accepting this cavalier approach to loops; and, pragmatically, whether unrolling is feasible, and will enable us to find more bugs, the basic goal of testing.

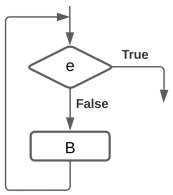

Fig. 1. Control flow for a loop

The rest of this article develops answers to these questions. Section 2 introduces a mathematically precise definition of loop unrolling. Section 3 explains how we added an automatic loop unrolling mechanism to an existing framework for generating tests automatically, relying on a combination of test and proof techniques. Section 4 analyzes and evalutes the results. Section 5 reviews existing work, in particular in an area that is distinct from testing but closely related to it: model checking, which has introduced the unrolling-like notion of *bounded* checking. Section 6 lists threats to validity and open issues. Finally, section 7 presents conclusions that current results suggest as to the suitability of adding loop unrolling to testing strategies and branch coverage measures.

2 A mathematical definition of loop unrolling

2.1 The need for a theoretical analysis

The notion of loop unrolling is intuitively clear: when we need (for example for testing purposes) a finite approximation for a loop **until** `e` **loop** `B` **end**, with its potentially infinite set of possible executions, use a set of programs that execute `B` not at all, once, twice, three times and so on.

In many cases this intuitive view is correct. For example with a loop computing the maximum of a non-empty array `a` indexed from 1 to `N`, **from** $\mathtt{M} := -\infty$; $\mathtt{i} := 1$ **until** $\mathtt{i} > \mathtt{N}$ **loop** $\mathtt{M} := \mathtt{max}\ (\mathtt{M}, \mathtt{a}\ [\mathtt{i}])$; $\mathtt{i} := \mathtt{i} + 1$ **end**, executing the body `k` times for $\mathtt{k} < \mathtt{N}$ will yield the maximum of the array slice at indexes 1 to `k`, which is

indeed an approximation of the final result. But sometimes this notion of approximation is far less clear. Assume for example that `M` is a positive integer, possibly large, and consider the loop **from** `x := −M; i := 1` **until** `x > 0` **loop** `x := −M + i; i := −2∗i` **end**. It yields `x` = $-M + 2^j$ for the smallest odd integer `j` such that $log_2(j) > M$. For lesser values of `j`, however, the value of `x` fluctuates widely, further off from the result (for even values) than the original approximation `M`! Unlike with the previous example, iterating the loop body an even number of times, less than the final number, does not give us an "approximation" of the result in any intuitive (as opposed to theoretical) sense.

More generally, we should not let ourselves be fooled by the view (informally OK, but not literally true) that "executing the loop **until** `e` **loop** `B` **end** means executing `B` 0 times, or 1 time, or any number of times". Depending on the details of the loop, certain numbers of iterations — such as, say, 4 iterations — may not be possible at all. A better formulation talks about executing the body *some* number of times (not *all* possible numbers below the maximum if any). The rest of this section develops the mathematical theory providing the precise framework removing any ambiguity or potential confusion.

The theory's underpinnings come from classic work in denotational semantics [23] (see also [15] and, and [17]) and abstract interpretation [7]. The presentation is based on earlier work [18] which, however, did not involve full proofs of properties, let alone machine-supported ones. All the formal properties stated in the present article have been proved and machine-checked using the Isabelle theorem prover [22] and are publicly available. To facilitate cross referencing, every theorem stated below comes with a name, such as /Concat_station/ below, appearing in smaller font; the same name appears in the Isabelle files for the corresponding property and its proof. (Names in all upper case, such as `CONCAT_DEF`, also appear in the Isabelle files; they denote definitions and hence do not require proofs.)

2.2 Assumptions

We consider a loop L in the simple form given above: `until e loop B end`. The informal semantics is the usual one: execute the instruction `B` ("body") 0 or more times, stopping as soon as `e` (the "exit condition") holds. This discussion studies how we can — in particular for testing purposes — approximate L by a sequence of nested conditionals:

L_0 = **check False end** *--Inapplicable program*

L_{i+1} = **if** ¬ e **then** B; L_i **end** *--REC_DEF*

(Here an instruction **check** p **end**, where p is a Boolean property, has no effect if p has value true upon execution, and otherwise makes the entire program in which it appears inapplicable. It corresponds to **fail** in the trace set model of section 2.4. One can think of it in practice as causing a run-time crash but more abstractly it is simply an incorrect instruction. In the special case used here, **check False end**, defining L_0, is simply a program that is never applicable,

regardless of the input. Another name for **check** in some verification formalisms is **assume** [14]; **check False end** is essentially the same as Dijkstra's "abort" [9].)

Subsequent L_{i+1}, for $i \geq 0$, are defined for a growing set of possible inputs: those that, in each case, require at most i executions of `B` before rendering `e` **True**.

Why is it important to produce such a sequence of approximations? A number of applications exist, for example in compiler optimization, high-performance computing and software verification; the concrete impetus for the present study is to improve on branch coverage. True "path coverage" would imply covering any number of executions of `B`, which is impossible in the general case; branch coverage goes to the other extreme of restricting that number to zero and one. Unrolling provides an intermediate solution: "execute" the loop a variable number of times (not just one), tuning the unrolling level in accordance with the testing needs and the available computing resources (since a higher level requires more testing time).

2.3 Notation: traces and states

For the purpose of this discussion, the semantics of a program P is given by the set Traces (P) of its (finite) traces for any given input. In fact we identify P with its traces.

A trace x is a finite, non-empty sequence of program states written $<x_1, x_2, \ldots>$.

In this definition, a program state is defined by the values of the program variables (in a general sense, which for an object-oriented program will include the whole heap) as well as a program location (the indication of which instruction of the program an execution is currently at). Intuitively it corresponds to what you see if you stop the program during execution and look at what the debugger tells you.

If s is a state (an element in a trace) and e is an expression, $s[e]$ is the value of e in state s. For example if s is the state resulting from executing the instruction sequence $x := 2$; $y := 5$ the value of $s[x * y]$ is 10.

The i-th element (state), of a trace x is written x_i. Its length (number of elements) is written $|x|$. Since traces are non-empty, $|x| > 0$ and there always exists a first state, x_1, and a last state, written x_L. A trace is "stationary" if it has only one element, i.e. is of the form $<x_1>$. (In this case x_1 is the same as x_L.) A stationary trace corresponds to an empty execution, which leaves the state unchanged.

A concatenation operator using the symbol "+" is available on traces; for example, <m, n> + <n, o, p> is <m, n, o, p>. As this example suggests, x + y is defined if and only if $x_L = y_1$: the last element of the first operand must be the same as the first element of the second one. The common element (n in the example) appears only once in the concatenation. The intuition behind this rule is that concatenating two program traces only makes sense if the first program ends in a state from which the second one can take over.

Formally, $x + y$ is defined as the trace z of length $|x| + | y| - 1$ such that $z_i = x_i$ for $1 \leq i \leq |x|$ and $z_i = y_{i-|x|+1}$ for $|x| + 1 \leq i \leq |z|$.

The "+" operator is associative and may be used for more than two operands, as long as they satisfy the requirement that the final state of every operand is the same as the initial state of the next.

If x + y = z: *--/Concat_ assoc/*

- If x is stationary, then y = z; if y is stationary then x = z. *--/Concat_ station/*
- We say that x is a **prefix** of z, written x <= z (the relation is a partial order). If y is not stationary then x is a **proper** prefix and we write x < z. *--/Concat_ order/*
- We also say that z is an **extension** (resp. proper extension) of x. (y is a "suffix" of z but we do not need that notion.)

A **test** is a condition — in other words, a Boolean expression — on states.

A trace x **satisfies** a test v if $s[v]$ holds for some state s in x. (Remember that a state includes a program location.)

Theorem: if x satisfies v and $x \leq z$, then z satisfies v. *--/Extension_ stable/*

2.4 Trace sets

Instructions and programs (in the underlying programming language) will be defined by their **trace sets**. A trace set is what the name indicates: a set of traces.

skip is the set of stationary traces (those of the form $<x_1>$, with just one state).

fail is the empty trace set. (Note that traces themselves cannot be empty, as they always have an initial state and a final state —- which are the same for a stationary trace — but a trace set can be empty.)

We say that a trace set A **tests** c, or is **a test of** c, if it contains a trace satisfying c.

If A and B are trace sets, A + B, also written A ; B, is the set of traces

$\{z \mid \exists x : \mathrm{A}, y : \mathrm{B} \mid \mathrm{Z} = x + y\}$ *--[CONCAT_ DEF]*

In other words, the set of all traces obtained by concatenating a trace from A and a trace from B. Unlike the "+" operator on traces, the "+" operator on trace sets is defined for any operands. (Non-concatenable trace pairs in A and B, meaning pairs such that $x_L \neq y_1$, simply do not yield any element of A + B.)

Theorems:

fail = **fail** ; A *--/Concat_ fail1/*

= A ; **fail** *--/Concat_ fail2/*

A = A ; **skip** *--/Concat_ skip1/*

= **skip** ; A *--/Concat_ skip2/*

The "≤" and "<" operators between traces similarly extend to trace sets: A ≤ B is defined as

$\forall x : \mathrm{A} \mid \exists y : \mathrm{B} \mid x \leq y$ *--And similarly for "<"*

Unlike the operator on traces, "$\leq$" on trace set is not an order relation since it is not antisymmetric.

We can "slice" trace sets by conditions, pre- and post-. If c is a Boolean expression and A is a set of traces:

Restriction: only retain traces whose initial state satisfies c

$\mathrm{c} \mathbin{/} \mathrm{A} \triangleq \{x\colon \mathrm{A} \mid x_1\ [\mathrm{c}]\}$ *--[RESTRICT_DEF]*

Corestriction: only retain traces whose last state satisfies c

$\mathrm{A} \setminus \mathrm{c} \triangleq \{x\colon \mathrm{A} \mid x_L\ [\mathrm{c}]\ \}$ *--[CORESTRICT_DEF]*

Since we define program semantics by traces, we may use the following notations for programs:

$\mathrm{A} \equiv \mathrm{B} \coloneqq \mathrm{Traces}\ (\mathrm{A}) = \mathrm{Traces}\ (\mathrm{B})$ *--[TRACESET_EQ]*

$\mathrm{A} \cup \mathrm{B} \coloneqq \mathrm{P}$ such that $\mathrm{Traces}\ (\mathrm{A}) \cup \mathrm{Traces}\ (\mathrm{B}) = \mathrm{Traces}\ (\mathrm{P})$ *--[TRACESET_UN]*

$\mathrm{A} \subseteq \mathrm{B} \coloneqq \mathrm{Traces}\ (\mathrm{A}) \subseteq \mathrm{Traces}\ (\mathrm{B})$ *--[TRACESET_SUB]*

$x \in \mathrm{A} \coloneqq x \in \mathrm{Traces}\ (\mathrm{A})$ *--[TRACESET_MEMB]*

2.5 Properties of trace sets

The following theorems (all checked mechanically) express formal properties of trace sets and other basic mechanisms introduced above.

$\mathrm{False} \mathbin{/} \mathrm{A}$	$= \mathbf{fail}$	*--/False_restrict/*
$\mathrm{True} \mathbin{/} \mathrm{A}$	$= \mathrm{A}$	*--/True_restrict/*
$\mathrm{A} \setminus \mathrm{False}$	$= \mathbf{fail}$	*--/False_corestrict/*
$\mathrm{A} \setminus \mathrm{True}$	$= \mathrm{A}$	*--/True_corestrict/*
$\mathrm{c} \mathbin{/} (\mathrm{d} \mathbin{/} \mathrm{A})$	$= (\mathrm{c} \wedge \mathrm{d}) \mathbin{/} \mathrm{A}$	*--/Two_restrict/*
$(\mathrm{A} \setminus \mathrm{c}) \setminus \mathrm{d}$	$= \mathrm{A} \setminus (\mathrm{c} \wedge \mathrm{d})$	*--/Two_corestrict/*
$(\mathrm{A} \setminus \mathrm{c})\ ;\ (\mathrm{d} \mathbin{/} \mathrm{B})$	$= (\mathrm{A} \setminus (\mathrm{c} \wedge \mathrm{d}))\ ;\ \mathrm{B}$	*--/Corestrict_restrict1/*
	$= \mathrm{A}\ ;\ ((\mathrm{c} \wedge \mathrm{d}) \mathbin{/} \mathrm{B})$	*--/Corestrict_restrict2/*
	$\subseteq \mathrm{A}\ ;\ \mathrm{B}$	*--/Corestrict_restrict3/*
$(\mathrm{A} \setminus \mathrm{c})\ ;\ (\neg\ \mathrm{c} \mathbin{/} \mathrm{B})$	$= \mathbf{fail}$	*--/Corestrict_restrict4/*
$(\mathrm{A} \setminus \mathrm{c})\ ;\ \mathrm{B}$	$= \mathrm{A}\ ;\ (\mathrm{c} \mathbin{/} \mathrm{B})$	*--/Corestrict_restrict5/*
$(\mathrm{v} \mathbin{/} \mathrm{A})\ ;\ \mathrm{B}$	$= \mathrm{v} \mathbin{/} (\mathrm{A}\ ;\ \mathrm{B})$	*--/Restrict_compose/*
$\mathrm{A}\ ;\ (\mathrm{B} \setminus \mathrm{v})$	$= (\mathrm{A}\ ;\ \mathrm{B}) \setminus \mathrm{v}$	*--/Compose_corestrict/*
$\mathrm{v} \mathbin{/} (\mathrm{A} \cup \mathrm{B})$	$= (\mathrm{v} \mathbin{/} \mathrm{A}) \cup (\mathrm{v} \mathbin{/} \mathrm{B})$	*--/Restrict_union/*
$(\mathrm{A} \cup \mathrm{B}) \setminus \mathrm{v}$	$= (\mathrm{A} \setminus \mathrm{v}) \cup (\mathrm{B} \setminus \mathrm{v})$	*--/Corestrict_union/*
$\mathrm{A}\ ;\ (\mathrm{B} \cup \mathrm{C})$	$= (\mathrm{A}\ ;\ \mathrm{B}) \cup (\mathrm{A}\ ;\ \mathrm{C})$	*--/Compose_union1/*
$(\mathrm{A} \cup \mathrm{B})\ ;\ \mathrm{C}$	$= (\mathrm{A}\ ;\ \mathrm{C}) \cup (\mathrm{B}\ ;\ \mathrm{C})$	*--/Compose_union2/*

If t tests A and $\mathrm{A} \leq \mathrm{B}$, then t tests B. *--/Test_leq/*

We will now use these concepts to define programming constructs and what they test.

2.6 Defining control structures

The standard program instructions and control structures are easy to express as trace sets.

We have already seen **skip** (defined as the set of one-element traces) and **fail** (defined as the empty trace set).

Sequencing (block structure) has also been defined already through the operator "+" or its equivalent ";", which corresponds to the use of this symbol of programming languages. Here it correspondingly concatenates traces.

2.7 Conditional instructions

We define the conditional instruction as

if v **then** A **end** $\triangleq$ (¬ v / **skip**) ∪ (v / A) *--[COND_DEF]*

For the present discussion we will not need the commonly used version of the conditional instruction including an **else** part, but adding it is trivial.

The definition corresponds to the intuitive semantics of conditionals: an execution of C does nothing if v has value False, and otherwise is an execution of A.

2.8 The power operator

To define loops in the present formalism, it is useful first to introduce an intermediate mechanism, the repetition, or "power operator", applicable to instructions. If `A` is an instruction, A^i denotes A iterated i times (**skip** for $i = 0$). The precise definition is by induction:

$A^0 \triangleq$ **skip** *--[POWER_BASE]*

$A^{i+1} \triangleq (A ; A^i)$ *--[POWER_STEP]*

2.9 Loops

We define the loop instruction L, written in programming language notation in the form **until** e **loop** B **end**, as

$$L \triangleq \bigcup_{i:\ \mathbb{N}} (\neg e \ /\ B)^i \ \setminus \ e \qquad \text{--[LOOP\_DEF1]}$$

This definition corresponds to the intuitive semantics of loops: an execution of L consists of 0 or more executions of `B`, from states in which `e` does not hold, such that the last of them produces a state where `e` holds.

Since the definition is a union, we can equivalently replace each element by the union of the preceding ones:

$$L = \bigcup L_i \qquad \textit{--[LOOP\_DEF2]}$$

where

$$L_i \triangleq \bigcup_{j<i} (\neg e \ / \ B)^j \setminus e \qquad \textit{--[DEF2\_Li]}$$

L_i describes executions that achieve `e` by executing `B` repeatedly, but (strictly) less than i times. In particular, L_0 is an empty set, meaning **fail**, and

L_1 = **skip** \ e *--/Loop_Skip1/*
= e / **skip** *--/Loop_Skip2/*

We can also express the L_i sequence (the sequence whose union of all terms defines L) inductively as

L_0 $\triangleq$ **fail** *--/Loop3_L0/*
L_{i+1} $\triangleq$ $L_i \cup ((\neg\ e\ /\ B)^i \setminus e)$ *--/Loop3_Li/*

2.10 A loop as a recursive conditional

One way to look at the loop L = "**until** `e` **loop** `B` **end**" is as a solution to the fixpoint equation

L = **if** ¬ `e` **then** `B`; L **end** *--[FIXEQUA_1]*

Rather than proving directly that loops as defined above (through the sequence L_i) satisfy [FIXEQUA_1], we consider the following sequence of programs inspired by this equation:

$\underline{L}_0$ $\triangleq$ **fail** *--[FIXDEF_BASE]*
$\underline{L}_{i+1}$ $\triangleq$ **if** ¬ e **then** B; $\underline{L}_i$ **end** *--[FIXDEF_STEP]*
= (e / **skip**) ∪ (¬ e / (B; $\underline{L}_i$)) *--by [COND_DEF]*
--/Fixdef2_step/

It yields a proposed alternative definition L for loops:

$$\underline{L} \triangleq \bigcup_{i:\mathbb{N}} \underline{L_i} \qquad \textit{--[LOOP\_DEF3]}$$

As a reminder, the original definition was (after adaptation)

$$L = \bigcup L_i \qquad \textit{--[LOOP\_DEF2]}$$

with L_i defined by [DEF2_Li] above.

2.11 The two views are equivalent

We will now prove that the definitions are equivalent, by showing by induction that $L_i = \underline{L}_i$ for all i.

Both L_0 and $\underline{L}_0$ are **fail**. Then for $i \geq 0$: *--/Def_equiv/*

$$
\begin{aligned}
L_{i+1} &= \bigcup_{j\le i}(\neg e/B)^j \backslash e && \text{--by [DEF2\_ Li]}\\
\underline{L}_{i+1} &= (e/\mathbf{skip}) \cup (\neg e/(B;\underline{L}_i)) && \text{--by /Fixdef2\_ step/}\\
&= (e/\mathbf{skip}) \cup (\neg e/(B;L_i)) && \text{--by induction hypothesis}\\
&= L_1 \cup (\neg e/(B;L_i)) && \text{--by /Loop\_ Skip2/}\\
&= L_1 \cup (\neg e/(B;\bigcup_{j<i}((\neg e/B)^j \backslash e))) && \text{--by [DEF2\_ Li]}\\
&= L_1 \cup ((\neg e/B);\bigcup_{j<i}((\neg e/B)^j \backslash e)) && \text{--by /Restrict\_ compose/}\\
&= L_1 \cup \bigcup_{j<i}(\neg e/B);((\neg e/B)^j \backslash e) && \text{--by /Compose\_ union1/}\\
&= L_1 \cup \bigcup_{j<i}((\neg e/B);(\neg e/B)^j) \backslash e && \text{--by /Compose\_ corestrict/}\\
&= L_1 \cup \bigcup_{j<i}((\neg e/B)^{j+1}) \backslash e && \text{--by [POWER\_ STEP]}\\
&= L_1 \cup \bigcup_{1\le j\le i}((\neg e/B)^j) \backslash e && \text{--Change of index}\\
&= L_1 \cup (L_i - L_1) && \text{--by [DEF2\_ Li]}\\
&= L_i && \text{--("—" is set difference)}\\
& && \text{--QED}
\end{aligned}
$$

Theorem: $L_i \subseteq L$ for every i. (This is also true of $\underline{L}_i$ since it is the same as L_i.) --/Under_ approx/

2.12 Some consequences

We call $\underline{L}_i$ the **i-unrolling** of the loop L. It is of the form

```
L_i ≜ if not e then
        if not e then
          B
          if not e then
            ...
              if not e then
                B
                check False end        --Corresponds to fail
              end
            ...
          end
        end
      end
```

with exactly i occurrences of B (i.e. if $i = 0$ the instruction fails, if $i = 1$ it executes B once or fails, if $i = 2$ it executes B once or twice or fails etc.). In the general case, an i-unrolling executes B at most i times if it can do so with 'e' each time not satisfied, and otherwise fails.

For every trace x of L, there is a smallest i such that x is a trace of $\underline{L}_i$. By the definitions, x is also a trace of $\underline{L}_j$ for all $j > i$. (Recall that $\underline{L}_i \subseteq \underline{L}_j$.) As a

consequence, for every test t of L, there is a minimum i such that t is a test of the i-th unrolling (and all subsequent ones). This development gives the theoretical framework that we need to unroll loops in the present work's testing strategy. The default unrolling level is 1 (we treat a loop like an **if ... then ... end**). The more we unroll, the more extensive the tests will be.

A "bug" is a test for a specific condition (an incorrectness condition). Note that since an i+1-unrolling includes all the traces, and hence all the bugs, of an i-unrolling, the number of bugs found by a test can only be an increasing function of the unrolling level. (Otherwise, there is something wrong with the implementation of the strategy.)

3 Implementation: adding loop unrolling to an automated test generation strategy

An automatic test generation strategy called "seeding contradiction (SC)", introduced by Huang et al. in [10] (using ideas also applied in other work combining proofs and tests, such as [20]), allows generating test suites that achieve full branch coverage. It relies on the AutoProof tool, a program proving framework internally based on the Boogie prover and an SMT solver such as Z3 [2, 8]. When generating tests for a loop, the SC inserts a faulty clause in the form of "`check false end`" (as highlighted in Figure 2) inside the loop body. When the loop body contains no conditional statement (a plain block), it injects a contradiction clause at the beginning of the loop body (Figure 2 (a)); if there are multiple branches inside the loop body, it injects a contradiction clause in each of the branches (Figure 2 (b)).

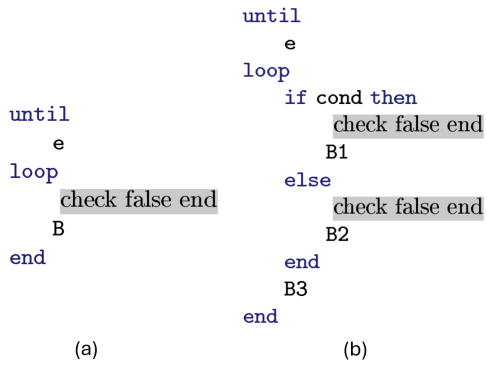

Fig. 2. Seeding contradictions for a loop

After verifying the seeded version, a program prover will report failures of the seeded clauses (as the assertions will always fail). It obtains a set of counterexample models from the underlying SMT solver, from which it produces executable test cases. Executions of those test cases are guaranteed to go through the locations of the injected "contradictions" and thus cover different branches in the

loop body. The SC strategy, however, can only ensure that the generated tests will enter the loop body, without guaranteeing the number of iterations the tests will perform. To produce tests that will explore the behaviors of the loop to a certain level, we extend the SC strategy by incorporating loop unrolling. We call the extended SC strategy SCU — seeding contradiction with unrolling. To allow generating tests that traverse the loop body a specific number of times, SCU performs instrumentation on the code with the loop unrolled to a certain level.

Figure 3 and 4 show the SCU approach. If the loop body is plain, SCU inserts a clause "`check not e end`" at the end of each unroll level $i \in \{1, ..., n\}$ (n is the loop unrolling factor). Adding such an assertion at level i forms a task for the prover — to find a counterexample for the property "`not e`" at the end of level i. If such a counterexample exists, SCU produces a test from the counterexample. During the execution of the test, the exit condition "`e`" holds at the end of level i, which enforces the loop to exit. In other words, the test is guaranteed to exercise the loop exactly i times.

Verification of the seeded version results in n failures and thus n tests; one for each unroll level. Note that some of the unrolled levels might not be reachable. For example, when a loop traverses an array whose size should be less than 10, the loop body is unreachable at unrolled level 10 or above. In those cases, the prover will produce no tests for the unreachable levels.

```
if not e then -- Unroll level 1
    B
    check not e end
    ...
    if not e then -- Unroll level i
        B
        check not e end
    end
    ...
end
```

Fig. 3. SCU for plain loop body

If the loop body contains conditionals, for each unrolling level i, SCU produces a test suite consisting of tests that go through every branch. Figure 4 shows the instrumentation of the unrolled loop, whose body contains two branches. SCU uses an integer variable `bn` to distinguish different branches in the unrolled loop. Let m be the number of branches in the original loop (here $m = 2$), the value of `bn` is in the range $[1, m * n]$. For each unroll level i, it identifies different branches by inserting at each of the branch an assignment "`bn` $:= j$", where $j \in [m * (i - 1) + 1,\ m * i]$. A unique value j identifies each branch. At the end of each unroll level, it inserts m assertions in the form of "`check not (e and bn = j)`". This assertion assigns a task to the prover to find a counterexample that will satisfy the property "`e and bn = j`". The test produced from the counterexample is guaranteed to go through the j^{th} branch and estab-

lish the exit condition at the end of unroll level i. If all branches are reachable at every unroll level, verification of the instrumented version results in $m * n$ tests.

```
if not e then -- Unroll level 1
    if cond then
        bn := 1
        B1
    else
        bn := 2
        B2
    end
    B3
    check not (e and bn = 1) end
    check not (e and bn = 2) end
    .. .
    if not e then -- Unroll level i
        if cond then
            bn := 2i − 1
            B1
        else
            bn := 2i
            B2
        end
        B3
        check not (e and bn = 2i − 1) end
        check not (e and bn = 2i) end
    end
    .. .
end
```

Fig. 4. SCU for conditionals inside loop body

4 Evaluation

We evaluate the implementation of SCU and discuss the trade-offs between performance and unrolling depth, with the objective to answer the following research questions:

- **RQ1** *What is the precise impact of loop unrolling on test generation time?*
- **RQ2** *What is the precise impact of loop unrolling on test execution time?*
- **RQ3** *Does loop unrolling actually lead to test suites that find more bugs?*

4.1 Experiment design

The experiment uses 12 examples that contain loops, adapted from examples in the AutoProof tutorial[1] and benchmarks of previous software verification competitions [25] [4] [12]. Each example contains exactly one loop. Table 1 lists their characteristics, including implementation size (number of Lines Of Code), number of branches in the loop body.

Table 1. Examples

Example	Lines of Code	#Branches in the loop
BINARY_SEARCH	67	3
MAX_IN_ARRAY	54	2
SQUARE_ROOT	55	3
FACTORIAL	41	0
GCD (Greatest Common Divisor)	128	2
SUM_AND_MAX	57	2
PRIME_CHECK	53	2
LINEAR_SEARCH	45	0
ARITHMETIC_ADD	49	0
ARITHMETIC_MULTIPLY	34	0
ARITHMETIC_DIVIDE	32	0
INVERSE	46	2

The experiment applies SCU to generate tests for those routines, with loops unrolled to different depths from 1 to 15. It then compares the fault-identification performance of the resulting test suites. For each of the examples, the experiment creates different faulty variants by randomly injecting errors into the correctly verified version. To assess the overall performance of each unrolling group, it then performs for each group 20 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. All code and results are available at `https://github.com/icst-2025-88/loop_unrolling`.

4.2 Analyses and results

Impact on Test Generation Time (RQ1) To answer RQ1: *What is the precise effect of loop unrolling on test generation time?*, we measure the test generation time, including the time for verification and for generating test scripts from counterexamples. Table 2 shows the test generation time of SCU when different unrolling depths are applied. In some cases, test generation or executing the generated tests would become intractable when unrolling depth exceeds a certain level: the experiment can only handle `BINARY_SEARCH` and `SQUARE_ROOT` up to unroll level 10, and `GCD` and `PRIME_CHECK` up to unroll level 8.

When the depth is below 5, a test generation task costs less than 1 seconds for most of the examples. For those examples with plain loop body (there is no conditional inside the loop), including `FACTORIAL`, `LINEAR_SEARCH`, `ARITHMETIC_ADD`, `ARITHMETIC_MULTIPLY`, `ARITHMETIC_DIVIDE`, the overall overhead roughly grows linearly as the unrolling depth increases. The time cost remains at the same scale. In the other 7 examples with branches inside the loop body, the time cost increases gradually at small depths, but the increment becomes more substantial

[1] http://autoproof.sit.org/autoproof/tutorial

as the depth becomes larger. The difference of the time cost between the initial and the final unrolling depths is significant and occurs on different scales. Those examples contain at least 2 branches in their loop bodies, resulting in the addition of more contradictory contracts during test generation by SCU.

Table 2. Test generation time

Example	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BINARY_SEARCH	0.95	0.63	0.80	0.77	0.99	1.33	1.46	1.96	2.77	7.05	–	–	–	–	–
MAX_IN_ARRAY	0.18	0.22	0.34	0.38	0.59	0.67	0.90	1.06	1.32	1.70	2.11	2.48	2.94	3.36	5.54
SQUARE_ROOT	0.10	0.21	0.30	0.53	0.53	1.16	1.67	2.97	3.85	6.41	–	–	–	–	–
FACTORIAL	0.38	0.10	0.11	0.13	0.13	0.15	0.16	0.19	0.21	0.24	0.27	0.30	0.33	0.35	0.40
GCD	0.09	0.11	0.18	0.39	0.61	0.90	1.70	4.27	–	–	–	–	–	–	–
SUM_AND_MAX	0.18	0.32	0.40	0.50	0.68	0.86	1.10	1.50	1.90	2.46	2.90	3.53	3.80	4.59	6.33
PRIME_CHECK	0.68	0.10	0.12	0.14	0.19	0.21	0.26	0.46	–	–	–	–	–	–	–
LINEAR_SEARCH	0.13	0.12	0.16	0.17	0.19	0.19	0.22	0.25	0.26	0.30	0.28	0.38	0.31	0.38	0.35
ARITHMETIC_ADD	0.08	0.11	0.13	0.16	0.17	0.19	0.21	0.25	0.27	0.29	0.34	0.36	0.40	0.42	0.46
ARITHMETIC_MULTIPLY	0.09	0.10	0.12	0.14	0.16	0.19	0.21	0.24	0.26	0.29	0.33	0.37	0.37	0.45	0.45
ARITHMETIC_DIVIDE	0.07	0.09	0.11	0.13	0.18	0.19	0.22	0.25	0.27	0.30	0.40	0.38	0.44	0.67	0.54
INVERSE	0.14	0.20	0.29	0.40	0.51	0.62	0.79	0.97	1.12	1.27	1.48	1.68	2.0	2.32	2.57

Impact on Test Execution Time (RQ2) To answer RQ2: *What is the precise effect of loop unrolling on test execution time?*, we measure the execution time of tests generated with different unrolling depths. Table 3 displays the average execution time for the tests of each unrolling group across the 20 runs. Overall, the test execution time increases linearly with the unrolling depth, remaining relatively small — under 0.5 seconds in most cases. The test execution time for `FACTORIAL` is notably high, as its contracts rely on a recursive function, the evaluating the correctness of the contracts incurs additional time cost. The increment in test execution time for `BINARY_SEARCH` is more substantial than the others, as it involves an array whose size grows exponentially with the unrolling depth. When the unrolling depth reaches 8, the size of the input array becomes considerably large, requiring more computational resources and time. This results in a significant increase in the execution time when depth rises from depth 8 to 10.

Table 3. Test execution time

Example	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BINARY_SEARCH	0.09	0.13	0.13	0.14	0.15	0.19	0.24	0.44	1.04	3.04	–	–	–	–	–
MAX_IN_ARRAY	0.01	0.02	0.02	0.03	0.04	0.05	0.05	0.06	0.07	0.08	0.09	0.09	0.10	0.11	0.12
SQUARE_ROOT	0.01	0.01	0.02	0.03	0.04	0.04	0.05	0.06	0.06	0.07	–	–	–	–	–
FACTORIAL	5.99	9.55	13.06	16.65	20.16	23.72	27.26	30.87	34.46	38.14	41.78	45.32	48.94	52.58	56.22
GCD	0.01	0.01	0.02	0.03	0.04	0.04	0.05	0.06	–	–	–	–	–	–	–
SUM_AND_MAX	0.01	0.01	0.02	0.03	0.04	0.04	0.05	0.06	0.07	0.07	0.08	0.09	0.10	0.11	0.11
PRIME_CHECK	0.01	0.02	–	0.02	0.03	0.04	0.05	0.06	–	–	–	–	–	–	–
LINEAR_SEARCH	0.01	0.02	0.02	0.03	0.04	0.05	0.06	0.07	0.07	0.08	0.09	0.10	0.11	0.12	0.13
ARITHMETIC_ADD	0.01	0.01	0.02	0.03	0.04	0.05	0.06	0.06	0.07	0.08	0.08	0.09	0.10	0.11	0.12
ARITHMETIC_MULTIPLY	0.01	0.02	0.03	0.04	0.05	0.06	0.06	0.07	0.08	0.09	0.10	0.11	0.12	0.13	0.14
ARITHMETIC_DIVIDE	0.01	0.02	0.02	0.03	0.04	0.05	0.06	0.06	0.07	0.08	0.09	0.10	0.11	0.11	0.12
INVERSE	0.01	0.02	0.02	0.03	0.04	0.05	0.05	0.06	0.07	0.08	0.09	0.09	0.10	0.11	0.12

Effectiveness of SCU (RQ3) To answer 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:

$$< program\ variant,\ tag\ of\ failed\ contract,\ line\ number >$$

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

- N_p: the number of distinct faults detected per run, which can be defined as a function:
$$N_p(i) = \frac{\sum_{j=1}^{20} |F_c(i,j)|}{20}$$
where $F(i,\ j)$ is the set of distinct faults found at the j^{th} run ($1 \leq j \leq 20$) with unrolling depth i. $N_p(i)$ represents the average performance of fault detection of unrolling group i. $N_p(i)$ 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.
- N_a: the number of all distinct faults that appear during the 20 repetition runs of that group, which can be described as a function over the unrolling depth i:
$$N_a(i) = |\bigcup_{j=1}^{20} F_c(i,\ j)|$$
Informally, $N_a(i)$ 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.

Table 4 and Table 5 display the results of N_p and N_a of the 12 examples. `Total` sums up the number of faults of all the examples. Overall, the execution of the generated tests detect 251 distinct faults during the experiment.

The result a significant improvement in fault detection as unroll depth increase: an average test suite produced by SCU is able to find 63.7% ($N_p(1) = 159.75$) of all distinct faults with unrolling depth of 1, while unrolling the loop 5 times effectively uncovers over 80% ($N_p(5) = 202.85$) of all faults. Considering all 20 runs, tests generated at unroll depth 1 is able to find 69.3% ($N_a(1) = 174$) of all faults; unrolling the loop 5 times is good enough to detect 96.4% ($N_a(5) = 242$) of the faults.

For most of the examples (10 out of 12), increasing the unrolling depth indeed helps in finding more faults. The benefit is most significant for `BINARY_SEARCH`, `GCD` `SUM_AND_MAX`, `SQUARE_ROOT`, and `FACTORIAL`. For some examples, the benefits brought by unrolling is minimal, typically resulting in only 1 or 2 additional

faults detected; those examples include `MAX_IN_ARRAY`, `ARITHMETIC_MULTIPLY`, `ARITHMETIC_DIVIDE`, `INVERSE`, `PRIME_CHECK`. In the cases of `LINEAR_SEARCH` and `ARITHMETIC_ADD`, however, unrolling appears to have no impact, with no additional faults detected as the unrolling depth increases. Both examples involve straightforward operations within the loop: `LINEAR_SEARCH` iterates through elements, while `ARITHMETIC_ADD` performs simple addition; they only alter the value of just a single variable. The result suggests that loops with more complex conditional statements or intricate data dependencies are more likely to benefit from unrolling. In contrast, loops with simple operations like basic summation or iterating through array elements see little advantage from unrolling; the faults in these cases seem identifiable without requiring deeper unrolling.

Table 4. Performance of bug detected per run (N_p)

Example	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BINARY_SEARCH	25.5	31.7	35.55	38.1	39.15	39.95	41.05	41.8	42.25	42.8	–	–	–	–	–
MAX_IN_ARRAY	4.9	5.25	5.45	5.45	5.5	5.5	5.55	5.55	5.55	5.6	5.65	5.7	5.8	5.8	5.8
SQUARE_ROOT	16	18	18.85	18.95	18.95	18.95	18.95	18.95	19	19	–	–	–	–	–
FACTORIAL	10	17	17	17	17	17	17	17	17	17	18	18	18	18	18
GCD	13	19	19.8	19.9	19.95	20	20	20	–	–	–	–	–	–	–
SUM_AND_MAX	8.1	13.35	13.55	14.25	15.1	16.45	16.5	16.65	16.85	17.25	17.4	17.8	17.85	18.15	18.15
PRIME_CHECK	22	22	22	22	22	22	22	22.55	–	–	–	–	–	–	–
LINEAR_SEARCH	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
ARITHMETIC_ADD	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
ARITHMETIC_MULTIPLY	9	9.5	10	10	12	12	12	12	12	12	12	12	12	12	12
ARITHMETIC_DIVIDE	10.5	11.5	11.5	11.5	12	12	12	12	12	12	12	12	12	12	12
INVERSE	19.75	19.95	20	20.1	20.2	20.2	20.3	20.5	20.6	20.6	20.6	20.6	20.8	20.8	20.9
Total	159.75	188.25	194.7	198.25	202.85	205.05	206.35	208	–	–	–	–	–	–	–

Table 5. Performance of bug detected over all runs (N_a)

Example	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BINARY_SEARCH	34	51	59	64	69	71	72	74	74	74	–	–	–	–	–
MAX_IN_ARRAY	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6
SQUARE_ROOT	16	19	19	19	19	19	19	19	19	19	–	–	–	–	–
FACTORIAL	10	17	17	17	17	17	17	17	17	17	18	18	18	18	18
GCD	13	19	20	20	20	20	20	20	–	–	–	–	–	–	–
SUM_AND_MAX	9	16	16	20	20	22	22	22	22	22	22	22	22	22	22
PRIME_CHECK	22	22	22	22	22	22	22	23	–	–	–	–	–	–	–
LINEAR_SEARCH	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
ARITHMETIC_ADD	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
ARITHMETIC_MULTIPLY	13	14	14	14	14	14	14	14	14	14	14	14	14	14	14
ARITHMETIC_DIVIDE	11	12	12	12	12	12	12	12	12	12	12	12	12	12	12
INVERSE	20	22	22	22	22	22	22	22	22	22	22	22	22	22	22
Total	174	219	228	237	242	246	247	250	–	–	–	–	–	–	–

Fig. 6 and Fig. 5 depicts the changes of faults detected in percentage when the unroll depth increases. This percentage is computed by dividing N_p and N_a by their maximum values appear during all testing sessions. The result shows that in most cases, when the depth is small (less than 5), the curves rise rapidly, suggesting significant improvements in fault detection. The most significant improvement occurs when the depth increases from 1 to 2, which results in an improvement of N_p by 11.3% and N_a by 17.9%. As the depth exceeds 5, the effect of unrolling on fault detection diminishes. Increasing the depth from 5 to 8 yields only a modest improvement of 2.1% for N_p and 3.6% for N_a, both of which are less substantial than the gains observed at small depths.

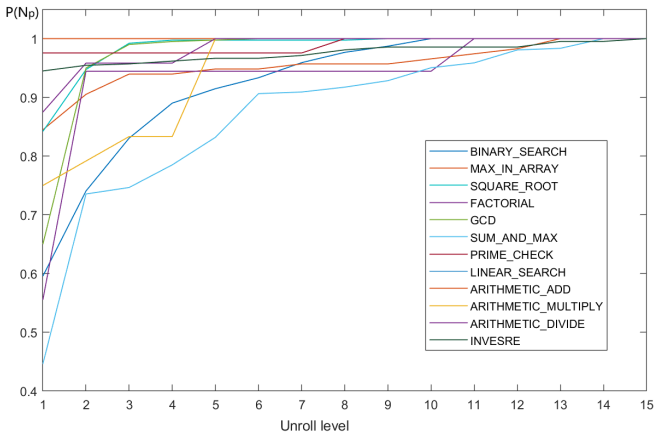

Fig. 5. $P(N_p)$: the percentage of faults detected per run at different unrolling levels.

The results presented, 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 automated test generation methods like SC, incorporating unrolling as an essential feature seems crucial. Further research and validation would be beneficial to explore this assertion more thoroughly.

5 Related work

One of the software verification technique that embodies the concept of loop unrolling is Bounded Model Checking (BMC) [3], which reduces model checking of linear temporal logic (LTL) formulas to propositional satisfiability. BMC operates by unrolling the transition relation of a finite state machine for a fixed number of steps k, and then checking whether a property violation can occur. If

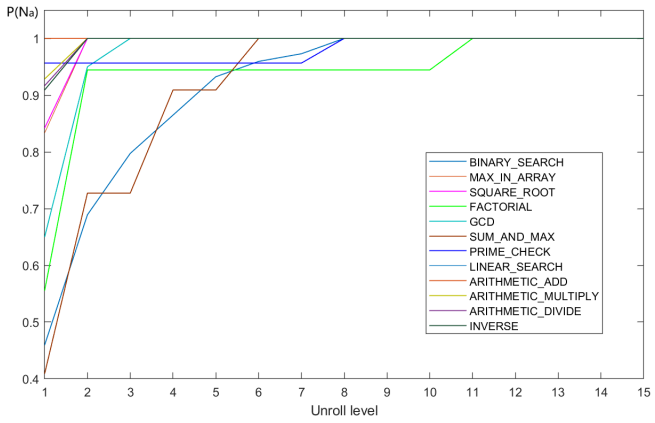

Fig. 6. $P(N_a)$: the percentage of faults detected over all 20 runs at different unrolling levels.

no violation is found, k is increased, and the process is repeated. This approach allows for a systematic exploration of the state space, with the bound k serving as a parameter to control the depth of the search.

The approach of this article, while similar in its unrolling technique, constructs different properties: if a prover finds a counterexample to an assertion, this counterexample serves as a test case, guaranteed to reach both the desired depth in the loop and the specified position within the loop body. Another distinctive property of the present work lies in its treatment of unwinding correctness. In the BMC literature, the correctness of unwinding is often assumed, either treated as self-evident or accepted axiomatically. The research reported here goes beyond this assumption: we provide a formal proof demonstrating the equivalence between the unrolled loop and the original loop structure.

Two notable tools that implement BMC are CBMC [13] and JBMC [5], which verify C and Java programs against the annotated assertions, with loops unrolled to a given depth. They can also be used as test generation tool, which automatically generate tests that satisfy a certain code coverage criteria. This concept of loop unrolling has also been integrated into a test generation tool PathCrawler [26, 27] which performs unit testing of C programs to obtain path coverage. It reduces the problem of covering all loop paths in a loop is to a *k-path* objective with the aim of covering loop paths within k loop iterations. Compared to the present work, such tools apply the idea of loop unrolling to either improve the efficiency of exploration of systems' behaviors or to systematically cover different loop paths.

Like the present work, Huster et al. [11] went beyond the traditional criterion of branch coverage and proposed an approach to detect more possible failures by explicitly addressing various patterns of loop iteration orders. They group iteration orders that influence one another into equivalence classes based on how

the current loop iteration affects the next, thereby reducing the complexity of covering all possible loop path variations.

6 Threats to validity and open issues

A limitation of the present work is the small size of the sample set of programs, and the small size of these programs themselves. Although some of them are real software elements (extracted from widely used libraries), they are not representative of large-scale production programs. They do, however, include significant loops, some of them sophisticated, and so provide a credible basis for studying the potential effects of unrolling.

Another issue is that many of the bugs (although not all) are seeded, rather than being actual bugs found in released code.

Also limiting the generalization of the present results is the use of an automated verification framework, AutoProof and the associated "seeding contradiction" test-generation framework of Huang et al., which at this stage is still a research tool rather than a deployed production environment.

While the results obtained in the experiments reported above seem strongly to suggest that loop unrolling may be feasible without an undue effect on testing time, they do not clearly uncover a "magic unrolling number" — an absolute constant `N` which would enable us to give a general rule-of-thumbs advice to practicing software developers, as in "unroll 5 times and you will be OK most of the time". Looking at Fig. 6 and Fig. 5 does suggest that something around `N = 5` would make sense, but one would need numerous experiments on large and representative code examples before such an initial heuristic would have enough confirmation to warrant inclusion into standard industry guidelines. We hope that the present work provides a solid basis for performing such experiments leading to firm empirical conclusions.

7 Conclusion

This paper has pursued both a theoretical aim and a practical one. The theoretical contribution is to provide a simple and sound mathematical theory of loop unrolling, avoiding the ambiguities and confusions that may result from a purely informal approach; all the corresponding properties come with a mechanically-checked proof with a leading proof tool, Isabelle. The practical contribution is to apply this theory to generate loop-unrolled test suites for a number of examples, of which some involve sophisticated loops. With the qualifications given in the previous section, the results indicate that: (1) Standard branch coverage, by ignoring loop body repetitions, *does* miss a significant number of bugs. (2) It *is* practically possible to correct this deficiency by adding automatically unrolled versions of loops to a test suite. (3) For small unrolling levels, the time penalty is reasonable, and justified by the potential for finding extra bugs. (4) The clear bug-finding outcomes confirm the usefulness of using a formal verification framework and applying it to generate high-coverage test suites, as exemplified in

recent research efforts at the frontier of tests and proofs, two complementary techniques of program verification.

We hope that these steps provide important information on an essential, if often overlooked, issue of software engineering: is the cavalier attitude that the industry commonly applies to loops, by ignoring the property that actually *defines* the concept of loop (the ability to repeat instructions!), justified — and should we not do better?

References

1. Ammann, P., Offutt, J., Huang, H.: Coverage criteria for logical expressions. In: 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003. pp. 99–107. IEEE (2003)
2. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB Standard: Version 2.0. In: International Workshop on Satisfiability Modulo Theories. vol. 13, p. 14 (2010)
3. Biere, A.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003), `https://api.semanticscholar.org/CorpusID:6692303`
4. 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. Springer (2011)
5. Brenguier, R., Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: A Bounded Model Checking Tool for Java Bytecode. arXiv:2302.02381 (2023)
6. Chekam, T.T., Papadakis, M., Le Traon, Y., Harman, M.: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 597–608. IEEE (2017)
7. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
8. 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. Springer (2008)
9. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
10. 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. Springer (2023)
11. Huster, S., Burg, S., Eichelberger, H., Laufenberg, J., Ruf, J., Kropf, T., Rosenstiel, W.: Efficient testing of different loop paths. In: Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings. pp. 117–131. Springer (2015)
12. Klebanov, V., Müller, P., , et al.: The 1st Verified Software Competition: Experience Report. In: International Symposium on Formal Methods (FM). pp. 154–168. Springer (2011)
13. 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. Springer (2014)
14. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR). pp. 348–370. Springer (2010)

15. Meyer, B.: Introduction to the theory of programming languages. Prentice-Hall (1990), publicly available version (updated 2023) at https://bertrandmeyer.com/ITPL.
16. Meyer, B.: Seven principles of software testing. Computer **41**(8), 99–101 (2008)
17. Meyer, B.: Theory of Programs, pp. 159–189. Springer (2015). https://doi.org/10.1007/978-3-319-28406-4_6, `https://doi.org/10.1007/978-3-319-28406-4_6`, available at `https://se.inf.ethz.ch/~meyer/publications/proofs/top.pdf`.
18. Meyer, B.: A formal definition of loop unrolling with applications to test coverage. arXiv (mar 2024), `https://arxiv.org/abs/2403.08923`
19. Meyer, B., Arkadova, A., Kogtenkov, A.: The Concept of Class Invariant in Object-Oriented Programming. arXiv (preprint of article submitted for publication) (2022), `https://arxiv.org/abs/2109.06557`
20. 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. IEEE (2022)
21. Nilizadeh, A., Leavens, G.T., Păsăreanu, C.S., Le, X.B.D., Cok, D.R.: Does going beyond branch coverage make program repair tools more reliable? In: 2024 IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 281–292. IEEE (2024)
22. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
23. Stoy, J.E.: Foundations of denotational semantics. In: Bjøorner, D. (ed.) Abstract Software Specifications. pp. 43–99. Springer (1980)
24. Wei, Y., Meyer, B., Oriol, M.: Is Branch Coverage a Good Measure of Testing Effectiveness?, pp. 194–212. Springer (2012), available at `https://se.inf.ethz.ch/~meyer/publications/testing/coverage.pdf`
25. 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. Springer (2008)
26. Williams, N.: Towards Exhaustive Branch Coverage with PathCrawler. In: Int. Conference on Automation of Software Tests (AST). pp. 117–120. IEEE (2021)
27. Williams, N., Marre, B., Mouy, P., Roger, M.: Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: European Dependable Computing Conference. pp. 281–292. Springer (2005)