Engineering a Formally Verified Automated Bug Finder

Arthur Correnson
Dominic Steinhöfel
arthur.correnson@cispa.de
dominic.steinhoefel@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

ABSTRACT

Symbolic execution is a program analysis technique executing programs with symbolic instead of concrete inputs. This principle allows for exploring many program paths at once. Despite its wide adoption—in particular for program *testing*–little effort was dedicated to studying the semantic foundations of symbolic execution. Without these foundations, critical questions regarding the correctness of symbolic executors cannot be satisfyingly answered: Can a reported bug be reproduced, or is it a *false positive* (soundness)? Can we be sure to find *all bugs* if we let the testing tool run long enough (completeness)? This paper presents a systematic approach for engineering provably sound and complete symbolic execution-based bug finders by relating a programming language's operational semantics with a symbolic semantics. In contrast to prior work on symbolic execution semantics, we address the correctness of critical implementation details of symbolic bug finders, including the search strategy and the role of constraint solvers to prune the search space. We showcase our approach by implementing WiSE, a prototype of a verified bug finder for an imperative language, in the Coq proof assistant and proving it sound and complete. We demonstrate that the design principles of WiSE survive outside the ecosystem of interactive proof assistants by (1) automatically extracting an OCaml implementation and (2) transforming WiSE to PyWiSE, a functionally equivalent Python version.

CCS CONCEPTS

• **Theory of computation** → **Program verification**; *Operational semantics*; Higher order logic; *Automated reasoning*; • **Software and its engineering** → *Semantics*; **Software testing and debugging**; **Formal software verification**; **Software verification**; *Automated static analysis*.

KEYWORDS

Symbolic Execution, Testing, Program Verification, Symbolic Semantics, Proof Assistants

INTRODUCTION

> It is now two decades since it was pointed out that program testing may convincingly demonstrate the presence of bugs, but can never demonstrate their absence. After quoting this well-publicized remark devoutly, the software engineer returns to the order of the day and continues to refine his testing strategies, just like the alchemist of yore, who continued to refine his chrysocosmic purifications.
>
> *E.W. Dijkstra,* 1988 [14]

A bridge that works fine for a year but collapses during a windy day is considered a failure. A program that works fine for a year but crashes for a certain input is considered buggy. This ridiculous analogy reflects the reality of software engineering: While civil engineers carefully identify loads and associated stresses on the structures they build and create their designs accordingly, preventing tragic crashes upfront, "bugs" in software are most frequently reported by *users* only after deployment [30]. However, software engineers are in a fortunate position: They can stress-test the *final product* before it is shipped to users. This becomes an opportunity only if programmers possess adequate verification tools. The predominant verification discipline today is *testing*, which aims to find bugs that are *actionable* in the sense that running the program under test with an *input* provided by the testing tool results in an error state. For example, *evolutionary graybox fuzzers* (such as AFL++ [15]) mutate inputs from an initial population and add mutants covering new parts in the program. While modern fuzzers work exceptionally well in many cases, the effectiveness of evolutionary fuzzers strongly depends on the quality of the initial seed inputs [27]. Furthermore, they struggle to cover code guarded with complex constraints. Consider the Python program

```python
if x == hash((y, z)): fail()
```

Finding a triple of values satisfying the `if` condition randomly or by mutating previous inputs is extremely difficult. Indeed, Bundt et al. [9] demonstrated that graybox fuzzers are "mostly useless" for certain bugs. However, they state that integrating *symbolic execution* with graybox fuzzers to so-called *hybrid fuzzers* is *highly effective,* a result also confirmed by other studies (e.g., [8]).

Symbolic Execution (SE) builds on the principle of declaring some (or all) inputs to a program to be *symbolic*. While concrete execution maintains states mapping variables to values, SE uses *symbolic states* consisting of a *path condition* and a *symbolic store.*

Arthur Correnson and Dominic Steinhöfel

At *branching points* such as the `if` statement from above, SE generates multiple successor states distinguished by different constraints added to their path condition. As usual, *assignments* update the symbolic store, which can map to symbolic values. SE was conceived almost simultaneously as a bug finding [24] and program proving technique [10] and is still actively used in both areas. It is more popular in testing, though, due to its *precise nature:* Under *ideal assumptions*, SE yields no false positives and can be used in automated testing campaigns; on the other hand, program *proving* with SE requires expensive auxiliary specifications [4].

So, what are these "ideal assumptions?" First, SE is a *whitebox* technique: It requires access to the source code of the tested program, interpreting its structure to drive the search for bugs. While this permits better results, implementing a symbolic executor requires a precise understanding of the semantics of the targeted language. For example, a symbolic executor for Python programs must respect Python's integer *floor division* semantics: The division expression x // y corresponds to floor(x / y), where "/" is division on reals. Other interpretations can result in *false positives* (spurious bugs) or *false negative* (missed bugs). In the program

<div align="center">

`if x // y == -2: fail()`

</div>

a wrong interpretation of integer division might lead a symbolic bug finder to believe that $x = -5$, $y = 2$ triggers fail(). However, in Python, the result of -5 // 2 is −3, while in other languages such as Java it evaluates to −2.

Second, SE discards unreachable erroneous states by checking whether their path conditions are satisfiable. In our examples from above, it must decide if "x == hash((y, z))" or "x // y == -2" are consistent with the previously collected constraints. To that end, the SE engines use *constraint solvers* to determine whether a set of constraints is satisfiable (*SAT*) or unsatisfiable (*UNSAT*). However, instead of returning *SAT* or *UNSAT*, solvers frequently *time out* providing no answers; in that case, blindly reporting a bug could result in a *false positive.*

Consequently, an SE engine is more difficult to implement correctly than a fuzzer. While the semantics and correctness of SE have been studied in different contexts [13, 25, 28, 33], the correctness of symbolic *bug finders* remains an open research topic. In particular, to our knowledge, there exists no rigorous definition of what constitutes a trustworthy symbolic bug finder.

The main correctness criterion for testing tools is the "soundness of bugs" [18]: If the tool answers *Is there a bug in my program?* with *yes*, there must exist a bug witness (an actionable input). We call this property soundness for testing (*T-soundness*). *Fuzz testers* (short *fuzzers*) generate inputs that they feed to the program under test and report a bug if the program crashes. Such tools are T-sound by construction. However, T-soundness alone is an insufficient quality measure for bug finders. Indeed, a tester never reporting *any* bug is trivially T-sound. Thus, the *converse* of soundness is an additionally desirable property: If a bug witness exists, a tester should eventually answer the question *Is there a bug in my program?* with *yes*. This property, which we name completeness for testing (*T-completeness*), is generally out of reach for fuzzers. The best option is to estimate the *residual risk* that an ongoing fuzzing campaign would discover an error if it only ran for a bit longer [7].

Contrary, building a T-sound and T-complete symbolic executor *is* possible. In this paper, we propose a general approach to building provably sound and complete bug finders based on symbolic execution. We implemented a prototypical bug finder based on this approach in the Coq proof assistant together with an end-to-end proof of its correctness. To the best of our knowledge, ours is the first endeavor to explicitly specify the *meaning of correctness in the context of symbolic bug finders* in a proof assistant and to use *formal verification techniques* to prove the correctness of a bug finder.

Research Questions. Precisely, we answer the following main questions: (1) *What constitutes a reliable testing tool?*, and (2) *How can we engineer such a tool?* These questions *cannot be answered experimentally*. Following the terminology of Shaw [31], they belong to the categories "Method for Evaluation" (Question (1)) and "Method/Means of Development" (Question (2)) of software engineering research questions. Our answer to Question (1) is the precise definition of T-soundness and T-completeness. Addressing Question (2), we explain the characteristics of a T-sound and T-complete bug finder and expound a methodology to implement such a tool. Following this methodology, we implemented SE-based testing tools in Coq and Python. For the Coq prototype, we derived a formal, mechanized correctness proof.

Related Work. Kapus and Cadar [22] combine compiler testing techniques and different oracles to test three symbolic bug finders. They reported 20 distinct bugs "exposing a variety of important errors"—e.g., related to division. This is the only work on the correctness of (symbolic) bug finders (Question (1)) we are aware of. Other work mainly studies how SE can be *applied* for automated testing. The strongest oracle used by Kapus and Cadar, the "output oracle," compares the results of a concrete execution and a symbolic one following the same path. This oracle only works if the initial path condition uniquely describes the inputs used for concrete execution, but is, at the same time, "obfuscated" to prevent the executor from dropping to "concrete mode." When testing SE engines with less constrained inputs, such that multiple paths are explored, Kapus and Cadar use coarser oracles. In contrast, we developed a *verified* symbolic bug finder based on carefully designed semantic foundations in response to Question (2). Likely, Kapus and Cadar would have found more bugs when investing even more work on test cases and oracles; our bug finder is guaranteed to be bug-free.

In formal program verification, it is much more common to address the soundness of verification tools and their underlying theoretical underpinnings. The goal of formal verification is not to show the presence but *prove the absence* of bugs. Consequently, the main correctness criterion is the soundness of proofs (P-soundness): If a verifier answers *Is my program safe to execute?* with *yes*, then there exists a proof of the program's correctness. As with T-completeness, P-completeness is the converse of P-soundness. Since checking a complex program proof is arguably more complex then checking whether an input causes a bug, the formal verification community put significant effort into justifying their tools and foundations. For example, Abstract Interpretation (AI) [12] is a mathematical framework for designing and implementing correct static analysis. Following the AI framework, Jourdan et al. [21] proved the soundness of an abstract interpreter for C programs in the Coq proof assistant. This project is strongly related to the development

of CompCert [26], a C compiler also proven correct in Coq. Coq itself is based on an established meta logic; its kernel was proven correct [32].

Closing the circle to symbolic execution, there have been various efforts to show *P-soundness* of SE-based verifiers [1, 23, 34]. Furthermore, the semantics of SE was addressed in the past [13, 25, 28, 33] in a more general context. The symbolic semantics used in this paper is inspired by the semantics proposed by de Boer and Bonsangue [13]. Building on these foundations, we define soundness and completeness in the context of symbolic bug finding and provide a formally verified implementation.

Contributions. Following the CompCert approach [5, 29] to build verified program analyzers in a proof assistant, we develop a prototype of a formally verified symbolic bug finder in Coq. We formalize the concrete operational semantics of a small imperative programming language and derive a symbolic operational semantics inspired by de Boer and Bonsangue [13] (Sect. 1). Then, we extensively explore the relations between concrete and symbolic semantics by proving that SE is a sound and complete method for reachability analysis and various bug-finding tasks (Sect. 2). Building on these theoretical results, we derive a T-sound and T-complete symbolic bug finder (Sect. 3). In the course of this, we formally address several practical questions such as *path selection* and *search space pruning* using constraint solvers and their influence on *exhaustiveness* (i.e., T-completeness) and *precision* (i.e., T-soundness) of SE. All implementations, definitions, theorems, and proofs in this paper are realized in Coq. Thus, if our *specifications* convince you, you can *blindly trust the proofs.* We call our prototypical symbolic bug finder WiSE ("What is Symbolic Execution"). To demonstrate the practicability of our SE engine's design principles, we (1) extract an executable OCaml implementation fully automatically from the Coq code and (2) manually transform WiSE to PyWiSE, a WiSE implementation written in Python (Sect. 4). Both extracted bug finders can be invoked on the command line. To provide hands-on evidence that WiSE works—additionally to our formal proofs—we apply it to find bugs inserted into numeric algorithms.

1 CONCRETE AND SYMBOLIC SEMANTICS

We study symbolic execution along the example of IMP, a small imperative programming language with loops and integer arithmetic. In this section, we introduce the IMP language (Sect. 1.1) and describe its operational semantics (Sect. 1.2). Afterward, we equip IMP with a symbolic operational semantics following de Boer and Bonsangue [13] (Sect. 1.3). The symbolic semantics will be the formal basis to justify the correctness of our SE engine.

Notation. We use the following notational conventions to clarify the domains of identifiers in our formalizations.

- $p, q, \ldots$ for *programs*
- V for *concrete environments*, S for *symbolic stores*
- φ for *path conditions* or *LTL formulas*
- σ for *concrete states* $\langle V, p \rangle$, $\hat{\sigma}$ for *symbolic states* $\langle \varphi, S, p \rangle$
- α for *streams* (e.g., of symbolic states)

Statements	*stmt*	::=	`skip`
		|	`fail`
		|	*VAR* `=` *aexpr*
		|	*stmt* `;` *stmt*
		|	`while` *bexpr* `do` *stmt* `od`
		|	`if` *bexpr* `then` *stmt* `else` *stmt* `fi`
Expressions	*bexpr*	::=	`true` | `false`
		|	*bexpr* `(` `and` | `or` `)` *bexpr*
		|	`not` *bexpr*
		|	*aexpr* `(` `==` | `<=` | `<` | `>=` | `>` `)` *aexpr*
	aexpr	::=	*INT* | *VAR*
		|	*aexpr* `(` `+` | `−` `)` *aexpr*

Figure 1: Syntax of IMP

1.1 A Small Imperative Programming Language

IMP is an imperative programming language with assignments, while loops, and conditionals. Figure 1 contains a grammar of IMP's concrete syntax; *VAR* denotes a variable identifier and *INT* an integer. We integrate a `fail` statement signaling an error. Using `fail`, one can instrument source code to, e.g., mark supposedly unreachable sections or model run-time assertions. IMP does not natively support multiplication and division; these operators can be implemented in terms of addition and subtraction.

$$\langle V, x = e \rangle \rightarrow \langle V[x := [\![e]\!]_V], \mathtt{skip} \rangle$$

$$\frac{[\![b]\!]_V = \mathtt{true}}{\langle V, \mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi} \rangle \rightarrow \langle V, s_1 \rangle}$$

$$\frac{[\![b]\!]_V = \mathtt{false}}{\langle V, \mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi} \rangle \rightarrow \langle V, s_2 \rangle}$$

$$\frac{\langle V_1, s_1 \rangle \rightarrow \langle V_2, s_2 \rangle}{\langle V_1, s_1\ ;\ s_3 \rangle \rightarrow \langle V_2, s_2\ ;\ s_3 \rangle}$$

$$\langle V, \mathtt{skip}\ ;\ s \rangle \rightarrow \langle V, s \rangle$$

$$\frac{[\![b]\!]_V = \mathtt{true}}{\langle V, \mathtt{while}\ b\ \mathtt{do}\ s\ \mathtt{od} \rangle \rightarrow \langle V, s\ ;\ \mathtt{while}\ b\ \mathtt{do}\ s\ \mathtt{od} \rangle}$$

$$\frac{[\![b]\!]_V = \mathtt{false}}{\langle V, \mathtt{while}\ b\ \mathtt{do}\ s\ \mathtt{od} \rangle \rightarrow \langle V, \mathtt{skip} \rangle}$$

$$[\![x]\!]_V := V(x),\ x \in \mathrm{Vars}$$

$$[\![\mathtt{true}]\!]_V := \mathtt{true} \qquad [\![\mathtt{false}]\!]_V := \mathtt{false}$$

$$[\![e_1 + e_2]\!]_V := [\![e_1]\!]_V + [\![e_2]\!]_V \qquad [\![b_1\ \mathtt{and}\ b_2]\!]_V := [\![b_1]\!]_V \wedge [\![b_2]\!]_V$$

$$\cdots$$

Figure 2: Operational Semantics of IMP

$$\langle \varphi, S, x = e \rangle \rightarrow_{\text{sym}} \langle \varphi, S[x := [\![e]\!]_S], \text{skip} \rangle$$

$$\langle \varphi, S, \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi} \rangle \rightarrow_{\text{sym}} \langle \varphi \text{ and } [\![b]\!]_S, S, s_1 \rangle$$

$$\langle \varphi, S, \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi} \rangle \rightarrow_{\text{sym}} \langle \varphi \text{ and } (\text{not } [\![b]\!]_S), S, s_2 \rangle$$

$$\frac{\langle \varphi_1, S_1, s_1 \rangle \rightarrow \langle \varphi_2, S_2, s_2 \rangle}{\langle \varphi_1, S_1, s_1 \; ; \; s_3 \rangle \rightarrow_{\text{sym}} \langle \varphi_2, S_2, s_2 \; ; \; s_3 \rangle}$$

$$\langle \varphi, S, \text{skip} \; ; \; s \rangle \rightarrow_{\text{sym}} \langle \varphi, S, s \rangle$$

$$\langle \varphi, S, \text{while } b \text{ do } s \text{ od} \rangle \rightarrow_{\text{sym}} \langle \varphi \text{ and } [\![b]\!]_S, S, s \; ; \; \text{while } b \text{ do } s \text{ od} \rangle$$

$$\langle \varphi, S, \text{while } b \text{ do } s \text{ od} \rangle \rightarrow_{\text{sym}} \langle \varphi \text{ and } (\text{not } [\![b]\!]_S), S, \text{skip} \rangle$$

Figure 3: Symbolic Semantics of IMP

1.2 Concrete Operational Semantics

IMP programs are executed in an *environment* $V : \text{Vars} \rightarrow \mathbb{Z}$ representing the values assigned to each program variable. Execution states are pairs $\langle V, p \rangle$ of an environment and a program that remains to be executed. The small-step operational semantics for IMP (Fig. 2) is a relation between execution states. We use the notation $\langle V, p \rangle \rightarrow \langle V', p' \rangle$ to state that executing program p in environment V for one step produces a new environment V' together with a program p' that remains to be executed. The relation "$\rightarrow^*$" is the reflexive, transitive closure of "$\rightarrow$." We use the notation $[\![e]\!]_V$ to denote the value of the expression e in environment V. The operational semantics is presented in Fig. 2.

1.3 Symbolic Operational Semantics

Informally, a program is symbolically executed by using symbolic placeholders instead of concrete values as inputs. When a statement with different possible outcomes is reached (in IMP, this can be an `if` statement or a loop), *both* possibilities are independently explored in different execution branches. Symbolic execution thus produces in a *tree* representing all possible execution paths. Each tree node is labeled with a formula (called the *path condition*), which characterizes the condition on the program inputs that needs to be met for the program to reach the associated state.

More formally, we regard SE as a type of program execution respecting a *symbolic* operational semantics "$\rightarrow_{\text{sym}}$" (Fig. 3). In SE, symbolic *stores* $S : \text{Vars} \rightarrow expr$ take the role of environments. Contrary to the latter, symbolic stores map variables to symbolic expressions and not numeric values. The dual to concrete execution states are *symbolic states*, triples $\langle \varphi, S, p \rangle$ of a formula φ (the *path condition*), a *symbolic store* S, and a program to be executed p (the *program counter*). We extend the concrete evaluation of an expression e in a store V to a symbolic evaluation $[\![e]\!]_S$. The result of $[\![e]\!]_S$ is again a symbolic expression obtained by replacing variables in e with their associated expressions in S, if any.

The symbolic execution of a program p usually starts in the initial state $\langle \text{true}, id, p \rangle$, where $id := x \mapsto x$ is an "empty" symbolic store assigning to each program variable a symbol of the same name. The initial path condition `true` indicates that we make no assumptions

about the program inputs. We write $\langle \text{true}, id, p \rangle \rightarrow^*_{\text{sym}} \langle \varphi, S, p' \rangle$ to denote that the symbolic state $\langle \varphi, S, p' \rangle$ can be reached by symbolic execution of p. Figure 4 shows a symbolic execution tree for the execution of a conditional statement in the default initial state.
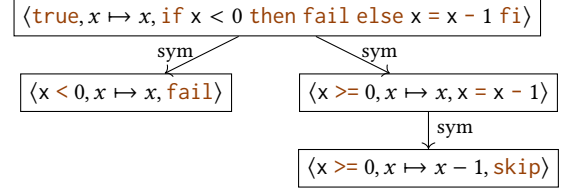
Figure 4: Example Symbolic Execution Tree

2 FINDING BUGS SYMBOLICALLY

A symbolic executor *simulates* many concrete executions of a program simultaneously. From this, we can obtain a bug finder by reporting an error whenever one simulated execution reaches an error state. Every reported bug must come with an *witness*. In fuzzing, this is a bug-triggering input; in the case of SE, it is frequently a more abstract witness in the form of a boolean formula describing *all* inputs triggering the bug. Additionally, we would like to ensure that any bug in the program will eventually be detected during SE. In this section, formally connect bug finding and symbolic execution. In particular, we lift the symbolic semantics from Sect. 1.3 to a sound and complete method to find bugs.

2.1 Sound & Complete Reachability

Our first step toward a symbolic bug finder is to connect symbolic and concrete executions. The glue we use to establish this connection is *reachability*. A correct symbolic executor should *only* reach symbolic states representing reachable concrete states. Conversely, we would like it to reach *any* reachable concrete state in the form of a symbolic state representing it.

We write $Reach(p, \sigma)$ and $Reach_{\text{sym}}(p, \sigma)$ to express that concrete state σ can be reached by concrete and symbolic execution of the program p. Then, we express soundness and completeness concerning reachability as a simple equivalence:

Theorem 2.1 (Soundness and Completeness for Reachability).
$$Reach_{sym}(p, \sigma) \Leftrightarrow Reach(p, \sigma)$$

$Reach(p, \sigma)$ can be expressed using the concrete operational semantics. A state σ is reachable for a program p if executing p in some initial store V_0 eventually leads to σ.

Definition 2.1 (Concrete Reachability).
$$Reach(p, \sigma) \triangleq \exists V_0, \langle V_0, p \rangle \rightarrow^* \sigma$$

Symbolic states reached during SE represent *many* concrete states. To make $Reach_{\text{sym}}(p, \sigma)$ precise, we relate them to their "concretizations." The concretization operation $\circ$ turns a symbolic store S into a concrete store $V' = V \circ S$ by replacing all free variables in S by their associated values in V. For example, if $S = \{x \mapsto y, y \mapsto y\}$ and $V = \{x \mapsto 1, y \mapsto 2\}$, we obtain $V' = \{x \mapsto 2, y \mapsto 2\}$.

Definition 2.2.
$$(V \circ S)(x) := [\![S(x)]\!]_V$$

We lift the *store* concretization "∘" to the concretization of symbolic *states*. To express that a symbolic state $\hat{\sigma}$ represents a concrete state σ via an initial store V_0, we write $\sigma \simeq_{V_0} \hat{\sigma}$. Any concretization σ of $\hat{\sigma}$ must have the *same program counter* as σ. Furthermore, the initial store V_0 must *satisfy the path condition* of $\hat{\sigma}$. Finally, σ's environment must be a concretization of $\hat{\sigma}$'s store via V_0.

DEFINITION 2.3.
$$\langle V, p \rangle \simeq_{V_0} \langle \varphi, S, p' \rangle \triangleq p = p' \wedge V = V_0 \circ S \wedge [\![\varphi]\!]_{V_0} = \text{true}$$

Now, all *concretizations of reachable symbolic states* are symbolically reachable. Formally:

DEFINITION 2.4.
$$Reach_{sym}(p, \sigma) \triangleq \exists V_0, \exists \hat{\sigma}, \langle \text{true}, id, p \rangle \rightarrow^*_{sym} \hat{\sigma} \wedge \sigma \simeq_{V_0} \hat{\sigma}$$

In the remainder of this section, we describe the proofs of both directions of Theorem 2.1 for the interested reader. Following the proofs, Sect. 2.2 addresses the detection of bugs, which involves making precise what constitutes an "error state."

PROOF OF SOUNDNESS

Subsequently, we derive the proof of "sound reachability," i.e., that all states reachable by SE are also reachable by concretely executing a program. This proof requires some intermediate lemmas. Lemma 2.1 is a compositionality result on the interplay of concretization and symbolic store updates.

LEMMA 2.1.
$$V \circ (S[x := [\![e]\!]_S]) = (V \circ S)[x := [\![e]\!]_{V \circ S}]$$

PROOF. By structural induction on e. □

During SE, path conditions get only ever *more specific* by accumulating addition constraints. This is asserted in Lemma 2.2. We note $\varphi \vDash \varphi' \triangleq \forall V, [\![\varphi]\!]_V = \text{true} \Rightarrow [\![\varphi']\!]_V = \text{true}$.

LEMMA 2.2.
$$\langle \varphi, S, p \rangle \rightarrow^*_{sym} \langle \varphi', S', p' \rangle \Rightarrow \varphi' \vDash \varphi$$

PROOF. By induction on the length of the $\rightarrow^*_{sym}$ derivation and by induction on the last $\rightarrow_{sym}$ step. □

Whenever the path condition after a symbolic step is satisfiable, we can make a corresponding concrete step.

LEMMA 2.3.
$$\langle \varphi, S, p \rangle \rightarrow_{sym} \langle \varphi', S', p' \rangle \wedge V \vDash \varphi' \Rightarrow \langle V \circ S, p \rangle \rightarrow \langle V \circ S', p' \rangle$$

PROOF. By induction over the symbolic derivation and by composition (2.1) in the case of variable assignments $x = e$. □

We extend Lemma 2.3 to the transitive reflexive closure by applying monotonicity (Lemma 2.2).

LEMMA 2.4.
$$\langle \varphi, S, p \rangle \rightarrow^*_{sym} \langle \varphi', S', p' \rangle \wedge V \vDash \varphi' \Rightarrow \langle V \circ S, p \rangle \rightarrow^* \langle V \circ S', p' \rangle$$

PROOF. By iterating Lemma 2.3 and by Lemma 2.2. □

Sound reachability follows from the (more general) Lemma 2.4.

THEOREM 2.2 (SOUND REACHABILITY).
$$Reach_{sym}(p, \sigma) \Rightarrow Reach(p, \sigma)$$

PROOF. By immediate application of Lemma 2.4 and the fact that $\langle V_0, p \rangle \simeq_{V_0} \langle \text{true}, id, p \rangle$. □

PROOF OF COMPLETENESS

We address the converse direction: All concretely reachable states are symbolically reachable. The idea is to show that for any execution step that can be taken according to the concrete semantics, there exists a symbolic step to simulate it.

LEMMA 2.5.
$$(\sigma \simeq_{V_0} \hat{\sigma} \wedge \sigma \rightarrow \sigma') \Rightarrow \exists \hat{\sigma}', \hat{\sigma} \rightarrow_{sym} \hat{\sigma}' \wedge \sigma' \simeq_{V_0} \hat{\sigma}'$$

PROOF. By induction on the derivation $\sigma \rightarrow \sigma'$ and selecting the appropriate symbolic rule for each case. □

We extend Lemma 2.5 to the reflexive transitive closure.

LEMMA 2.6.
$$(\sigma \simeq_{V_0} \hat{\sigma} \wedge \sigma \rightarrow^* \sigma') \Rightarrow \exists \hat{\sigma}', \hat{\sigma} \rightarrow^*_{sym} \hat{\sigma}' \wedge \sigma' \simeq_{V_0} \hat{\sigma}'$$

PROOF. By iterating Lemma 2.5. □

The completeness of $Reach_{sym}$ follows directly from Lemma 2.6.

THEOREM 2.3 (COMPLETE REACHABILITY).
$$Reach(p, \sigma) \Rightarrow Reach_{sym}(p, \sigma)$$

PROOF. It is easy to see that $\langle V_0, p \rangle \simeq_{V_0} \langle \text{true}, id, p \rangle$. The result immediately follows from this observation and Lemma 2.6. □

2.2 Sound & Complete Bug Finding

In the previous section, we proved that the states reachable by SE are the same as those reachable by concrete execution. Building on that, we establish that SE can also be used as a sound and complete bug-finding method. First, however, we must address the question *What does it mean to find bugs?* We present different views on this question and prove that SE answers each case.

Detecting Buggy Programs. A natural view on bug finding is to ask—and answer—the question *Does my program contain a bug?* We introduce a relation $HasBug(p)$ characterizing buggy programs. This predicate is central to our definition of T-soundness and T-completeness (Sect. 3.5). A *state* $\sigma = \langle V, p \rangle$ "has a bug" if p is *stuck*, i.e., has no semantic successor state. However (in the case of IMP), this would also hold if p was skip, although we were more thinking of fail. We explicitly exclude the skip case in the following definitions of "not stuck" (i.e., "can progress") and "stuck."

DEFINITION 2.5 (PROGRESS).
$$Progress(\langle V, p \rangle) \triangleq p = \text{skip} \vee \exists \sigma, \langle V, p \rangle \rightarrow \sigma$$

DEFINITION 2.6 (STUCK STATES).
$$Stuck(\sigma) \triangleq \neg Progress(\sigma)$$

A buggy program has a reachable, stuck state.

DEFINITION 2.7 (HAVING A BUG).
$$HasBug(p) \triangleq \exists \sigma, Reach(p, \sigma) \wedge Stuck(\sigma)$$

As the question *Does my program contain a bug?* can be reduced to reachability, we can answer it symbolically.

DEFINITION 2.8.
$$HasBug_{sym}(p) \triangleq \exists \sigma, Reach_{sym}(p, \sigma) \land Stuck(\sigma)$$

The equivalence of $HasBug_{sym}$ and $HasBug$ follows immediately from the equivalence of $Reach_{sym}$ and $Reach$.

THEOREM 2.4 (SOUND AND COMPLETE BUG DETECTION (1)).
$$HasBug_{sym}(p) \Leftrightarrow HasBug(p)$$

PROOF. By soundness of $Reach_{sym}$ in the "$\Rightarrow$" direction and by completeness of $Reach_{sym}$ in the "$\Leftarrow$" direction. □

Finding Erroneous Execution States. $HasBug(p)$ asserts that *some* bug exists. Alternatively, we can specifically ask whether a *given* state is erroneous and whether SE can find it. The predicate $IsBug(p, \sigma)$ asserts that σ is an error state of p. In Sect. 3.4, we derive a refined "relative" version of T-soundness and T-completeness for nonterminating symbolic executors using $IsBug(p, \sigma)$.

DEFINITION 2.9.
$$IsBug(p, \sigma) \triangleq Reach(p, \sigma) \land Stuck(\sigma)$$

Similarly to above, we derive a *symbolic* predicate $IsBug_{sym}(p, \sigma)$.

THEOREM 2.5 (SOUND AND COMPLETE BUG DETECTION (2)).
$$IsBug_{sym}(p, \sigma) \Leftrightarrow IsBug(p, \sigma)$$

Finding Erroneous Inputs. Our final view is to ask for "bad," i.e., bug-triggering, *inputs*. The predicate $BadInput(p, V_0)$ characterizes those (where the environment V_0 represents an input):

DEFINITION 2.10.
$$BadInput(p, V_0) \triangleq \exists \sigma, \langle V_0, p \rangle \to^* \sigma \land Stuck(\sigma)$$

The symbolic equivalent is more subtle since $\to_{sym}^*$ ranges over *symbolic* states. We must concretize the reached symbolic state before checking whether it denotes a bug.

DEFINITION 2.11.
$$BadInput_{sym}(p, V_0) \triangleq \exists \sigma \exists \hat{\sigma},$$
$$\langle true, S, p \rangle \to_{sym}^* \hat{\sigma} \land \sigma \simeq_{V_0} \hat{\sigma} \land Stuck(\sigma)$$

The equivalence of $BadInput_{sym}$ and $BadInput$ follows from the definition of $Reach_{sym}$ and the proof of Theorem 2.2 and Theorem 2.3.

THEOREM 2.6 (SOUND AND COMPLETE BUG DETECTION (3)).
$$BadInputs_{sym}(p, V_0) \Leftrightarrow BadInputs(p, V_0)$$

This view bridges the gap between SE and fuzzing: What is needed to report a *bug-triggering input* is (1) a bug *oracle* (a symbolic equivalent of $Stuck(\sigma)$) and (2) a constraint solver extracting a solution V_0 from a path constraint. Subsequently, we describe how to build a testing tool based on our insights on SE, including a brief discussion of "crash oracles." Constraint solving is not in our scope.

3 A TRUSTWORTHY SYMBOLIC BUG FINDER

Section 2 demonstrated the feasibility of finding bugs with SE based on the symbolic operational semantics. Now, we turn these insights into an executable correct symbolic bug finder. First, we discuss how to detect errors, resolve the nondeterminism of the symbolic semantics, and deal with diverging executions (Sect. 3.1). Second, we provide relevant implementation details of our WiSE prototype

(Sect. 3.2). Third, we introduce "relative" notions of soundness and completeness and relate them to the symbolic executor (Sect. 3.3) and bug finder (Sect. 3.4). Finally, we bridge the (already tiny) gap toward a T-sound and T-complete bug finder (Sect. 3.5).

3.1 From the Symbolic Semantics to an Executable Symbolic Executor

Our symbolic operational semantics is a solid basis for symbolic bug finding. Yet, its design leaves some crucial questions open that need to be addressed when developing an executable SE-based testing tool. These are: (1) The symbolic semantics is *nondeterministic*. Whenever an `if` or `while` statement is reached, SE can *choose* which branch to execute next. (2) We need an effective way to detect error states. The previously regarded predicates $HasBug_{sym}$, $IsBug_{sym}$, and $BadInput_{sym}$ are adequate formalizations for bug detection but are not executable because they live on the *semantic* level. Finally, (3) the symbolic semantics does *not terminate* for programs with loops, which we cannot solve in general, but must face nonetheless.

$$HasBug_{sym}(p)$$
$$\Leftrightarrow$$
$$\exists \sigma, Reach_{sym}(p, \sigma) \land Stuck(\sigma)$$
$$\Leftrightarrow$$
$$\exists \sigma, (\exists V_0, \exists \hat{\sigma}, \langle true, id, p \rangle \to_{sym}^* \hat{\sigma} \land \sigma \simeq_{V_0} \hat{\sigma}) \land Stuck(\sigma)$$
$$\Leftrightarrow$$
$$\exists V_0, \exists \langle \varphi, S, q \rangle,$$
$$\langle true, id, p \rangle \to_{sym}^* \langle \varphi, S, q \rangle \land \llbracket \varphi \rrbracket_{V_0} = true \land Stuck(\langle V_0 \circ S, q \rangle)$$
$$\Leftrightarrow$$
$$\exists \langle \varphi, S, q \rangle, \boxed{\langle true, id, p \rangle \to_{sym}^* \langle \varphi, S, q \rangle}_{(1)} \land$$
$$\boxed{\exists V_0, \llbracket \varphi \rrbracket_{V_0} = true}_{(2)} \land \boxed{Stuck(\langle V_0 \circ S, q \rangle)}_{(3)}$$

Figure 5: Decomposition of symbolic bug finding into checks for (1) reachability, (2) satisfiability, and (3) "stuckness."

Resolving Nondeterminism. While in theory, we could spawn independent processes whenever SE performs a case distinction when facing an `if` or `while` statement, this cannot continue forever. The number of branches in a symbolic execution tree is exponential in the number of such branching points; eventually, we will *have to* decide which branches to follow first. This process is known as *path selection* in SE [3]. Usually, path selection is regarded as a "heuristics," merely an artifact of efficiency considerations. However, the exploration order has solid consequences on the *T-completeness* of the resulting bug finder. Early dynamic symbolic executors such as DART [19] explored programs using a Depth-First Search (DFS). This approach is *incomplete* in general: Such executors likely get lost in ever deeper iterations of loops at the beginning of a program, neglecting any states that follow. Instead, we choose a Breadth-First Search (BFS) approach. In Sect. 3.3, we prove the completeness of BFS-based path selection, which requires estimating where a successor state is added in the stream of states to explore.

Detecting Errors. Any bug finder depends on a *bug oracle* to determine when a program state constitutes a bug. Predicates such as $HasBug_{sym}$ are non-executable *semantic* notions. Moreover, they are defined using symbolic reachability, which requires checking if a path condition is *satisfiable* (see Fig. 5). Fuzzers commonly classify program runs as erroneous if they end with a *crash* of the program. We use a similar approach: We classify a program as buggy or crashing if it is *stuck*. In our case, it can be shown that a state is stuck if and only the next statement to be executed is `fail`. This simple solution is already quite powerful. For example, to extend IMP with failing *expressions,* one can use `fail` to model the exceptional cases. Consider integer division: For any expression `x / y`, we add a preceding statement `if y == 0 then fail else d = div(x,y)`, replacing `x / y` with the fresh variable `d`. The function `div` would implement division using repeated subtraction, *assuming* that `y` is nonzero.

Addressing Termination. As a method simulating concrete executions, SE does not terminate for most looping or recursive programs. Derivations in our symbolic semantics (Fig. 3) are infinite for *any* program with a loop. In Sect. 3.5, we integrate pruning of unsatisfiable states, which solves that issue for loops with concrete or unsatisfiable guards. Even so, loops with underconstrained symbolic guards induce an infinite amount of reachable successor states.

There are two main options for dealing with nontermination in SE. The first option is to impose an upper bound on the number of times a loop is unwound. This approach has the advantage that it is easy to implement. However, it prevents us from reasoning about the *T-completeness* of a symbolic executor, since the chosen bound might be too small to detect sufficiently deep bugs. The approach we chose instead is to implement the symbolic executor as a generator of an *infinite stream* of states. A user of a stream-based executor can only ever inspect finitely many states from the stream, which corresponds to bounded execution; yet, we *can obtain a completeness result.* Our OCaml and Python implementations (Sect. 4) retrieve elements from the stream up to a user-specified bound.

Thus, we view automated bug finders as producers of lazily evaluated infinite sequences of status messages reporting on the progress of the bug finding process. There are various ways of implementing infinite streams depending on the capabilities of the programming language a symbolic executor is implemented in. In proof systems based on purely functional languages such as Coq, infinite sequences can be modeled by *co-induction* [16, 17]. In languages with lazy evaluation such as Haskell, such infinite sequences can be implemented using standard *lists*. In the case of eagerly evaluated functional languages such as OCaml, the implementation can resort to *explicit suspensions*. Otherwise, we can model streams with *generators à la* Python.

In Sect. 3.3, we show that the stream-based approach permits elegant specifications of correctness based on temporal logic.

3.2 Functional Implementation

After many dry discussions, we provide some implementation details of our proven-correct symbolic executor. As discussed in Sect. 3.1, the executor uses a breadth-first goal selection and produces a "lazy" infinite stream of status messages. We use an OCaml-flavored syntax in all code snippets; we found it easy to translate

```
let run l =
  match l with
  | [] ->
    yield None
    run []
  | ⟨φ, S, p⟩ :: l ->
    yield ⟨φ, S, p⟩
    run (l + expand ⟨φ, S, p⟩)
```

Figure 6: Main Entry Point of the Symbolic Executor

```
let expand ⟨φ, S, p⟩ =
  match p with
  | skip | fail -> [ ]
  | skip ; c ->
    [ ⟨φ, S, c⟩ ]
  | x = e ->
    [ ⟨φ, S[x := ⟦e⟧_S], skip⟩ ]
  | c_1 ; c_2 ->
    [ ⟨φ', S', c_1 ; c_2⟩ for ⟨φ', S', c_1⟩ in expand c_1 ]
  | while cond do c od ->
    [ ⟨φ and cond, S, c ; p⟩, ⟨φ and (and cond), S, skip⟩ ]
  | if cond then c_1 else c_2 fi ->
    [ ⟨φ and cond, S, c_1⟩, ⟨φ and (not cond), S, c_2⟩ ]
```

Figure 7: Computing Successors of Symbolic States

this syntax to, e.g., Python. Everything discussed in this section is implemented in Coq, proven correct, and automatically extracted to an executable OCaml executable version.

Symbolic Evaluator and Symbolic Stores. Implementing a symbolic executor requires an efficient implementation of symbolic stores and a function to evaluate symbolic expressions. One way to implement symbolic stores is using hash tables with variable names as keys and expressions as values. Updating a store S with an assignment $x := e$, written $S[x := e]$, returns an updated copy of the hash table. The symbolic evaluation $\llbracket e \rrbracket_S$ of an expression e in a store S can be performed by a simple recursive function. We abstract from these implementation details and use the mathematical notations $S[x := e]$ and $\llbracket e \rrbracket_S$ in code snippets to denote assignments and symbolic evaluations.

A Lazy Symbolic Executor. The main component of the symbolic executor is a function expand (Fig. 7) computing the successors of a symbolic state, implementing the symbolic semantics in Fig. 3. The executor's entry point is a function run (Fig. 6) taking a FIFO queue of symbolic states and returning a stream of (optional) reachable symbolic states. At every iteration, it yields the first state from the queue. Then, the direct successors of the current state are computed using expand and enqueued in the task queue (with the lowest priority, realizing a breadth-first search). Finally, run calls itself recursively on the updated queue. If the queue is empty, a None token is yielded indicating that no states are left for expansion.

```
let display state =
  if state is None then
    Finished
  else if is_stuck state then
    BugFound(state)
  else
    Pending

let bug_finder p =
    map display (run [ ⟨true, id, p⟩ ])
```

Figure 8: Status Message Conversion, Bug Finder Entry Point

```
let is_stuck ⟨φ, S, p⟩ =
  match p with
  | fail -> True
  | p ; _ -> is_stuck p
  | _ -> False
```

Figure 9: Classifying Error States

Reporting Bugs. The function run implements a symbolic executor. We obtain a bug finder by turning symbolic states from the stream into adequate *status messages:*

(1) BugFound($\langle \varphi, V, p \rangle$) informs the user that a bug has been detected and returns the associated symbolic state.
(2) Pending informs the user that the SE is in progress.
(3) Finished informs the user that the SE terminated. No further symbolic state will be discovered.

The conversion from symbolic states to status messages is accomplished by the function display (Fig. 8). The final bug finder (bug_finder in Fig. 8) starts SE with a singleton task queue consisting of an initial symbolic state and applies display on the results.

Detecting Bugs. The function display uses is_stuck (Fig. 9) to filter error states. As discussed in Sect. 2.2, is_stuck considers a symbolic state $\langle \varphi, S, p \rangle$ an error state if p starts with fail.

3.3 Relative Soundness and Completeness of the Symbolic Executor

T-soundness and T-completeness are expressed assuming a "yes/no" oracle. For example, a T-sound bug finder only responds with *yes* to *Is there a bug in my program?* if there *is* a bug that manifests for a concrete input. However, fuzzers and symbolic bug finders generally do not terminate unless interrupted, as in the case of our symbolic bug finder from the previous section. Thus, we adapt these notions to testers producing a *stream* of results. We call the resulting concepts *relative* soundness/completeness. They address the question of whether *an input in a stream exposes a bug.*

More precisely, a bug finder is *relatively sound* if any bug report in the generated stream corresponds to bug-exposing input(s). It is *relatively complete* if it will eventually report a bug for any given bug-triggering input. In other words, both terms are relative to the

time provided to the bug finder. Moreover, relative completeness is parametric in a concrete bug-triggering input. Though formally weaker than T-completeness due to the additional precondition, it provides the strong guarantee a *specific* bug will be found, not just *any* bug. In the context of SE, relative *soundness* additionally permits conveying false positives with *unsatisfiable path constraints;* excluding these requires a complete constraint solver that correctly classifies all unsatisfiable path conditions. Provided such a solver, it is an easy exercise to construct a T-sound and T-complete bug finder from a relatively sound/complete one, as described in Sect. 3.5.

In this section, we investigate SE-specific notions of relative soundness and completeness, asserting that a *symbolic executor* explores *only* reachable symbolic states (soundness) and *all* reachable states (completeness). The subsequent Sect. 3.4 formalizes the more general notions of relative soundness and completeness and applies the results from this section to prove that the bug finder from Sect. 3.2 satisfies these properties.

To conveniently express predicates over streams of symbolic states or status messages, we use Linear Temporal Logic (LTL) [2, Chapter 5] notation. LTL formulas describe infinite streams of elements of some carrier set A. Atomic LTL formulas are predicates over A-elements, or, equivalently, subsets of A. For example, the LTL formula $P = \{n \mid \exists m, 2 \cdot m = n\}$ is satisfied by the stream $4, 5, 6, \ldots$ since its *first* element is even. The formula $\Box P$ ("globally") describes streams in which *all* elements are even; $\Diamond P$ ("eventually") matches streams with *some* even number. The property $\Box \Diamond P$ holds for streams with infinitely many even numbers ("every position in the stream must eventually be succeeded by an even number"). In addition to "$\Box$" and "$\Diamond$," we use logical implication "$\rightarrow$." For example, $\{42\} \rightarrow (\Diamond \{0, 1\})$ means that any stream whose first element is 42 must contain either 0 or 1 at a later position.

We write $\alpha \vDash \varphi$ if the stream $\alpha = \alpha_0, \alpha_1, \ldots$ satisfies the LTL formula φ; $\alpha \nvDash \varphi$ means that α does *not* satisfy φ. Formally, the semantics of LTL formulas is inductively defined as follows:

$$
\begin{aligned}
\alpha \vDash P & \Leftrightarrow P(\alpha_0) \\
\alpha \vDash \Box \varphi & \Leftrightarrow \forall i \geq 0, \alpha_i \alpha_{i+1} \ldots \vDash \varphi \\
\alpha \vDash \Diamond \varphi & \Leftrightarrow \exists i \geq 0, \alpha_i \alpha_{i+1} \ldots \vDash \varphi \\
\alpha \vDash \varphi_1 \rightarrow \varphi_2 & \Leftrightarrow \alpha \nvDash \varphi_1 \text{ or } \alpha \vDash \varphi_2
\end{aligned}
$$

Our symbolic executor produces a stream of optional (possibly None) symbolic states from a list l of input states. To be sound, every symbolic state in the stream must be *reachable* from some state in l. The LTL formula *ReachableFrom*(l) asserts that the first element in a stream is None or a symbolic state reachable from a state in l.

DEFINITION 3.1 (REACHABLE FROM).
$\quad ReachableFrom(l) := \{\hat{\sigma}' \mid \exists \hat{\sigma} \in l, \hat{\sigma} \rightarrow^*_{sym} \hat{\sigma}'\} \vee \{\text{None}\}$

Our bug finder's soundness depends on the single-step execution function expand correctly implementing the relation $\rightarrow_{sym}$:

THEOREM 3.1 (CORRECT EXPANSION).
$\quad \hat{\sigma} \rightarrow_{sym} \hat{\sigma}' \Leftrightarrow \hat{\sigma}' \in \text{expand}(\hat{\sigma})$

PROOF. By induction on (1) the $\rightarrow_{sym}$-derivation (direction "$\Rightarrow$") and (2) the structure of $\hat{\sigma}$'s program counter ("$\Leftarrow$"). □

Now we can state the relative soundness theorem, whose proof follows directly from Theorem 3.1:

THEOREM 3.2 (RELATIVE SOUNDNESS OF SYMBOLIC EXECUTION).
$\text{run}(l) \vDash \Box ReachableFrom(l)$

By outputting a None value in the stream, run signals that all reachable states have been *exhaustively explored.* Consequently, the stream should only consist of None after the first issued one, guaranteeing that we can *safely* terminate SE after the first None.

THEOREM 3.3 (SOUND TERMINATION OF SYMBOLIC EXECUTION).
$\text{run}(l) \vDash \Box(\text{None} \rightarrow \Box \text{None})$

We consider the symbolic executor *relatively complete* if, for any initial symbolic state $\hat{\sigma}$, it generates *at least* all $\rightarrow^*_{\text{sym}}$-successors of $\hat{\sigma}$ when started on the list $[\hat{\sigma}]$. We first state a more general theorem (Lemma 3.1) from which we then conclude the relative soundness. Lemma 3.1 asserts that for any state $\hat{\sigma}$ discovered during SE, its (direct or indirect) successors will eventually be found. Its proof requires us to predict *when exactly* the direct successors of any state $\hat{\sigma}$ will be discovered by our breadth-first search strategy.

LEMMA 3.1 (REACHABILITY OF SUCCESSORS).
$\hat{\sigma} \rightarrow^*_{sym} \hat{\sigma}' \Rightarrow \forall l, \text{run}(l) \vDash \Box(\hat{\sigma} \rightarrow \Diamond \hat{\sigma}')$

PROOF. By induction on the length of the derivation $\hat{\sigma} \rightarrow^*_{\text{sym}} \hat{\sigma}'$. If $\hat{\sigma}' = \hat{\sigma}$, then the theorem is obvious. Otherwise, we have $s \rightarrow_{\text{sym}} \hat{\sigma}''$ and $\hat{\sigma}'' \rightarrow^*_{\text{sym}} \hat{\sigma}'$ for some intermediate state $\hat{\sigma}''$. Now suppose that $\hat{\sigma}$ occurs at a given position $i_{\hat{\sigma}}$ in the stream $\text{run}(l)$, the direct successor of $\hat{\sigma}''$ necessarily occurs at index $i_{\hat{\sigma}''} = i_{\hat{\sigma}} + |l| + 1$ or $i_{\hat{\sigma}''} = i_{\hat{\sigma}} + |l| + 2$ because run extends the task list by adding all the successors (see Theorem 3.1) of the current state at the end of the task list and there are at most two successors for any state. Since $\hat{\sigma}''$ occurs at position $i_{\hat{\sigma}''}$, we know by induction hypothesis that $\hat{\sigma}'$ occurs at some index $i_{\hat{\sigma}''} + n$ for $n \geq 0$. □

From Lemma 3.1, we conclude the completeness result.

THEOREM 3.4 (RELATIVE COMPLETENESS OF SYMBOLIC EXECUTION).
$\hat{\sigma} \rightarrow^*_{sym} \hat{\sigma}' \Rightarrow \text{run}([\hat{\sigma}]) \vDash \Diamond \hat{\sigma}'$

PROOF. It suffices to see that the first element of the stream $\text{run}([\hat{\sigma}])$ is $\hat{\sigma}$ and then to apply Lemma 3.1 at position 0. □

3.4 Relative Soundness and Completeness of the Symbolic Bug Finder

In the previous section, we established the relative soundness and completeness of the symbolic executor regarding the symbolic semantics. Now, we define *relatively sound bug discovery*, a property relevant to bug finders in general, including fuzzers. We assume the interface of the function find_bugs from Sect. 3.2. That is, a bug finder outputs a stream of status messages that are either Pending, Finished, or BugFound. For a status message to be *valid,* a BugFound message must report a symbolic state corresponding to a concrete set of bug-triggering states. The following definition of *ValidStatus* uses a function *Concrete*($\hat{\sigma}$) mapping from a symbolic state to the set of concrete states it represents. Formally, $Concrete(\hat{\sigma}) = \{\sigma \mid \exists V_0, \sigma \simeq_{V_0} \hat{\sigma}\}$.

DEFINITION 3.2 (VALID STATUS).
$ValidStatus(p) :=$
$\quad \{\text{BugFound}(\hat{\sigma}) \mid \forall \sigma, \sigma \in Concrete(\hat{\sigma}) \Rightarrow IsBug(p, \sigma)\}$
$\quad \vee \text{ Pending} \vee \text{Finished}$

According to this definition, a bug report is always valid for symbolic states with unsatisfiable path conditions since these have an empty *Concrete*($\hat{\sigma}$). How to get from there to real T-soundness is the focus of Sect. 3.5. The definition of relative soundness is easy: A relatively sound bug finder only outputs valid status messages.

THEOREM 3.5 (RELATIVELY SOUND BUG DISCOVERY).
$\text{find\_bugs}(p) \vDash \Box ValidStatus(p)$

PROOF. Direct application of Lemma 3.1, and the fact that display filters only stuck states. □

Completeness. Relative completeness means that any given bug will eventually be reported in the stream of status messages. As mentioned before, this property is both *weaker*—due to the precondition of providing a concrete bug-triggering input—and *stronger*—since it guarantees that a pre-chosen bug will be found—than T-completeness. The subsequent completeness theorem uses a function *Symbolic*(σ) mapping from a concrete state to the set of symbolic states representing it (the inverse of *Concrete*). More formally, $Symbolic(\sigma) = \{\hat{\sigma} \mid \exists V_0, \sigma \simeq_{V_0} \hat{\sigma}\}$.

THEOREM 3.6 (RELATIVELY COMPLETE BUG DISCOVERY).
$IsBug(p, \sigma) \Rightarrow \text{find\_bugs}(p) \vDash \Diamond Symbolic(\sigma)$

PROOF. Suppose $IsBug(p, \sigma)$. By completeness of bug finding, we have $IsBug_{\text{sym}}(p, \sigma)$. The result follows by the completeness of run and the fact that display keeps all stuck states in the stream. □

Termination. Implementing a complete and yet always *terminating* bug finder is impossible as it would solve the Halting problem. Yet, there are programs for which SE can be exhaustive (e.g., programs without loops nor recursion). We define "sound termination" as the property that whenever a bug finder emits a Finished message, it will not report anything else from that point on. If a soundly terminating, relatively complete bug finder does not report a bug and says Finished, we can be sure that *the tested program is safe.*

THEOREM 3.7 (SOUND TERMINATION OF THE BUG FINDER).
$\text{find\_bugs}(p) \vDash \Box(\text{Finished} \rightarrow \Box \text{Finished})$

PROOF. Direct consequence of Theorem 3.3. □

3.5 A T-Sound and T-Complete Bug Finder

A T-sound and T-complete bug finder is a *binary oracle* answering *yes* if, and only if, the tested program is faulty. So far, we considered bug finders producing infinite streams of status messages, to which T-soundness and T-completeness do not directly apply. Yet, relative soundness and completeness (Theorem 3.6, Theorem 3.5) are sufficiently strong to derive such an oracle. Relative soundness ensures for every BugFound($\hat{\sigma}$) message that all $\sigma \in Concrete(\hat{\sigma})$ are error states. The only obstacle in the way toward T-soundness is that *Concrete*($\hat{\sigma}$) could be empty, i.e., $\hat{\sigma}$'s path condition *unsatisfiable.* Otherwise, a BugFound message from the stream corresponds to a *sound* bug. SE engines typically use off-the-shelf constraint solvers such as Z3 or CVC5 to determine if a path condition is unsatisfiable.

Relative completeness, on the other hand, ensures that for any error state σ, the stream contains a message BugFound($\hat{\sigma}$) with $\hat{\sigma} \in Symbolic(\sigma)$. The latter condition implies that $\hat{\sigma}$'s path condition is satisfiable. Inconveniently, a *terminating* bug finder can

```
exception Bug
exception Termination

let report msg =
  match msg with
  | BugFound (path, _, _) ->
    if is_sat path then raise Bug
  | Finished -> raise Termination
  | _ -> ()

type answer = YES | NO

let has_bug p =
  try iter report (find_bugs p)
  with
    | Bug -> YES
    | Termination -> NO
```

Figure 10: A T-sound and T-complete Bug Finder

never be complete in general; the bug could always be hidden in the next state. Our solution to this problem is to implement a *nonterminating but complete* symbolic tester. Our implementation, shown in Figure 10, builds on `find_bugs` and uses a constraint solver `is_sat` and a function `iter` for iterating over all elements of a stream. It only outputs "sound" bugs, terminates for finite symbolic executions, and continues SE as long as no bug is found or the process is interrupted. Since only terminating functions can be implemented in Coq, we have to resort to a "pen-and-paper" proof of `has_bug`'s T-soundness and T-completeness.

THEOREM 3.8 (T-SOUNDNESS AND T-COMPLETENESS).
$$\text{has\_bug}(p) = YES \Leftrightarrow HasBug(p)$$

PROOF. The answer is YES iff there is a $\text{BugFound}(\langle \varphi, S, p' \rangle)$ message in the stream $\text{find\_bugs}(p)$ such that φ is satisfiable. The result follows from relative soundness/completeness. □

4 IMPLEMENTATIONS AND CASE STUDIES

In this section, we connect to Question (2) from the introduction: *How can we engineer a reliable symbolic testing tool?* We aim to propose a design for symbolic executors independently of the implementation language; for the solution to be of general interest, it should be able to "live" outside an interactive theorem prover. Thus, we derived *two executable implementations* from the Coq implementation of WiSE. First, we automatically extracted OCaml code, wrapped in a command line interface with a parser for IMP's concrete syntax. Second, we developed a functionally equivalent Python implementation, PyWiSE, again with a usable command-line frontend. We used Python generators to implement streams where the Coq version uses co-induction and re-implemented the Coq code with only small, purely syntactical changes.

Since the transformations from Coq to executable programs are not formally proven correct, we evaluated them with three numeric algorithms. We chose the computation of a number's factorial and integer square root and the greatest common divisor of

two numbers. We annotated each program with assertions of their correctness; e.g., for an integer square root r of a number n, n must be in the closed interval $[r^2; (r+1)^2]$. Next, we derived a "buggy" mutation of each program, e.g., turning `while _s > x do` into `while _s < x do` in the square root example. Finally, we asserted that our symbolic executors find the inserted bug but report no bug for the original programs. In addition, we verified that SE terminates with a `Finished` message for the correct programs when we restrict the domain of the input number(s) to a finite range. To demonstrate that a breadth-first path selection is superior to depth-first (as in the original DART symbolic executor [19]) in finding *any* bug in a program, we added a `--depth-first` option to PyWiSE to transition to depth-first search. As a result, the bug in the factorial implementation is not uncovered since SE gets "trapped" in infinite iterations of an early loop.

Engineering Tasks. The main novelty of this paper is the presentation of a *mechanized* specification of the correctness of testing tools and the implementation of a *mechanically verified* symbolic testing tool in a proof assistant. We abstracted the addressed problem in two noteworthy ways to facilitate this project.

(1) Our target programming language IMP is a *simple WHILE language.* Extending our framework to a richer language requires the specification of a richer semantics. For example, the semantics of ES5 JavaScript in the JSCert project [6] is implemented in about 3,000 lines of Coq code. The correctness proof of JSRef, an executable reference interpreter for JSCert, spans 3,500 lines of code.

(2) We disregard the question of *efficiency,* which is orthogonal to our Questions (1) and (2). Yet, efficiency is crucial for the competitiveness of SE-based testing tools compared to random testing [8]. An efficient symbolic executor might replace the breadth-first path selection with, e.g., a coverage-guided generational search [20]. A considerable body of literature on efficient symbolic execution exists (cf. the survey by Baldoni et al. [3] for an overview). The role of our contribution is to show how the correctness of an SE system can be retained in the face of efficiency optimizations. Exchanging path selection, for example, requires proving that all reachable symbolic states are *eventually* considered by the new selection mechanism (Lemma 3.1).

Extending our foundational work to efficient systems for richer languages is possible. Of course, that step requires significant (proof) engineering work. This paper shows the way.

5 CONCLUSION AND FUTURE WORK

Considering that testing is the predominant program verification technique, it may seem surprising that the correctness of bug finders has not received more attention. Probably, this is because fuzzers, the most popular automated testing tools, are naturally "bug-sound"—they only report inputs that *have already* made the program under test crash. However, fuzzers only find shallow bugs for programs with complex demands on their inputs or with code guarded by complex constraints. Symbolic Execution (SE) is a white-box technique simulating multiple program executions at once. As such, this technique can handle complex input constraints fully

automatically. Compared to a fuzzer, correctly implementing a symbolic executor constitutes a much more significant challenge. We defined when a tester is "bug-sound" and "bug-complete" (i.e., a program is buggy if, and only if, a bug is reported), resulting in the notions of T-soundness and T-completeness. We chose a semantics for SE and demonstrated that it can be used as a foundation for symbolic bug finding. Finally, we implemented a symbolic bug finder (WiSE) in Coq, proved it T-sound and T-complete, and extracted executable implementations in OCaml and Python.

The resulting symbolic executors interpret programs in the simple programming language IMP. Connecting to our foundational work, we plan to support additional IMP features, such as functions and pointers, in our symbolic executor. Thus, we could discover common pitfalls in implementing their symbolic evaluation. More generally, WiSE constitutes a solid basis for studying more advanced SE features, such as different path selection algorithms, constraint representations, or constraint-solving approaches.

Is it realistic to expect a fully verified symbolic executor for an industrial programming language? We think it is—considering the existence of fully verified *C compilers*. In any case, blackbox testing will never find all the errors in existing symbolic executors; but insights from implementing different programming language features in a framework will positively influence their design.

DATA AVAILABILITY

Our WiSE and PyWiSE prototypes and the documentation of the Coq sources can be accessed at

<center>

https://github.com/acorrenson/WiSE/

https://acorrenson.github.io/WiSE/

</center>

PyWiSE is also available on PyPI and can be installed via

<center>

`pip install wise-se`

</center>

An artifact with a Docker container comprising WiSE, PyWiSE, and our example programs in a working environment is publicly available [11].

ACKNOWLEDGMENTS

REFERENCES

[1] Wolfgang Ahrendt, Andreas Roth, and Ralf Sasse. 2005. Automatic Validation of Transformation Rules for Java Verification Against a Rewriting Semantics. In *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference (LPAR'05), Proceedings (Lecture Notes in Computer Science, Vol. 3835)*, Geoff Sutcliffe and Andrei Voronkov (Eds.). Springer, 412–426. https://doi.org/10.1007/11591191_29

[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. The MIT Press. http://www.amazon.com/Principles-Model-Checking-Christel-Baier/dp/026202649X%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D026202649X

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

[4] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. 2012. Lessons Learned From Microkernel Verification – Specification is the New Bottleneck. In *Proceedings Seventh Conference on Systems Software Verification, (SSV'12) (EPTCS, Vol. 102)*, Franck Cassez, Ralf Huuck, Gerwin Klein, and Bastian Schlich (Eds.). 18–32. https://doi.org/10.4204/EPTCS.102.4

[5] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. 2013. Formal Verification of a C Value Analysis Based on Abstract Interpretation. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer, 324–344.

[6] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 87–100. https://doi.org/10.1145/2535838.2535876

[7] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 230–241. https://doi.org/10.1145/3468264.3468570

[8] Marcel Böhme and Soumya Paul. 2016. A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Trans. Software Eng.* 42, 4 (2016), 345–360. https://doi.org/10.1109/TSE.2015.2487274

[9] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2021. Evaluating Synthetic Bugs. In *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security*, Jiannong Cao, Man Ho Au, Zhiqiang Lin, and Moti Yung (Eds.). ACM. https://doi.org/10.1145/3433210.3453096

[10] Rod M. Burstall. 1974. Program Proving as Hand Simulation With a Little Induction. In *Information Processing, Proceedings of the 6th IFIP Congress 1974*, Jack L. Rosenfeld (Ed.). North-Holland, 308–312.

[11] Arthur Correnson and Dominic Steinhöfel. 2023. Artifact for "Engineering a Formally Verified Automated Bug Finder". https://doi.org/10.5281/zenodo.8269801

[12] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 238–252.

[13] Frank S. de Boer and Marcello M. Bonsangue. 2019. On the Nature of Symbolic Execution. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11800)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer, 64–80. https://doi.org/10.1007/978-3-030-30942-8_6

[14] Edsger W. Dijkstra. 1988. *On the cruelty of really teaching computing science*. Center for American History, University of Texas at Austin. http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF EWD-1036.

[15] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[16] Eduardo Giménez. 1996. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In *Types for Proofs and Programs*, Stefano Berardi and Mario Coppo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 135–152.

[17] Eduardo Giménez and Pierre Castéran. 2007. A Tutorial on (Co-)Inductive Types in Coq. Available at https://www.labri.fr/perso/casteran/RecTutorial.pdf.

[18] Patrice Godefroid. 2005. The Soundness of Bugs is What Matters (Position Statement). In *PLDI'05 Workshop on the Evaluation of Software Defect Detection Tools (BUGS'05), Proceedings*. https://patricegodefroid.github.io/public_psfiles/bugs2005.pdf

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223. https://doi.org/10.1145/1065010.1065036

[20] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'08)*. The Internet Society. https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[21] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, Sriram K. Rajamani and David Walker (Eds.). ACM, 247–259. https://doi.org/10.1145/2676726.2676966

[22] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE

Computer Society. https://doi.org/10.1109/ASE.2017.8115669

[23] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified Symbolic Execution with Kripke Specification Monads (and no Meta-Programming). *Proc. ACM Program. Lang.* 6, ICFP (2022), 194–224. https://doi.org/10.1145/3547628

[24] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976). https://doi.org/10.1145/360248.360252

[25] Ralf Kneuper. 1991. Symbolic Execution: A Semantic Approach. *Sci. Comput. Program.* 16, 3 (1991), 207–249. https://doi.org/10.1016/0167-6423(91)90008-L

[26] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009). https://doi.org/10.1007/s10817-009-9155-4

[27] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. 2023. Reachable Coverage: Estimating Saturation in Fuzzing. In *45th IEEE/ACM International Conference on Software Engineering (ICSE'23), Proceedings.* ACM. https://mboehme.github.io/paper/ICSE23.Effectiveness.pdf to appear..

[28] Dorel Lucanu, Vlad Rusu, and Andrei Arusoaie. 2017. A Generic Framework for Symbolic Execution: A Coinductive Approach. *J. Symb. Comput.* 80 (2017), 125–163. https://doi.org/10.1016/j.jsc.2016.07.012

[29] Valentin Robert and Xavier Leroy. 2012. A Formally-Verified Alias Analysis. In *Certified Programs and Proofs (CPP 2012) (Lecture Notes in Computer Science, Vol. 7679).* Springer, 11–27.

[30] Rollbar. 2021. The State of Software Code Report. https://content.rollbar.com/hubfs/State-of-Software-Code-Report.pdf. Accessed: 2023-01-24.

[31] Mary Shaw. 2003. Writing Good Software Engineering Research Papers. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, Lori A. Clarke, Laurie Dillon, and Walter F. Tichy (Eds.). IEEE Computer Society, 726–737. https://doi.org/10.1109/ICSE.2003.1201262

[32] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2020. Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 8:1–8:28. https://doi.org/10.1145/3371076

[33] Dominic Steinhöfel. 2020. *Abstract Execution: Automatically Proving Infinitely Many Programs.* Ph. D. Dissertation. Darmstadt University of Technology, Germany. http://tuprints.ulb.tu-darmstadt.de/8540/

[34] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2015. Featherweight VeriFast. *Log. Methods Comput. Sci.* 11, 3 (2015). https://doi.org/10.2168/LMCS-11(3:19)2015