To Run What No One Has Run Before: Executing an Intermediate Verification Language[⋆]

Nadia Polikarpova, Carlo A. Furia, and Scott West

Chair of Software Engineering, ETH Zurich, Switzerland
`firstname.lastname@inf.ethz.ch`

Abstract. When program verification fails, it is often hard to understand what went wrong in the absence of concrete executions that expose parts of the implementation or specification responsible for the failure. Automatic generation of such tests would require "executing" the complex specifications typically used for verification (with unbounded quantification and other expressive constructs), something beyond the capabilities of standard testing tools.
This paper presents a technique to automatically generate executions of programs annotated with complex specifications, and its implementation for the Boogie intermediate verification language. Our approach combines symbolic execution and SMT constraint solving to generate small tests that are easy to read and understand. The evaluation on several program verification examples demonstrates that our test case generation technique can help understand failed verification attempts in conditions where traditional testing is not applicable, thus making formal verification techniques easier to use in practice.

1 Help Needed to Understand Verification

Static program verification has made tremendous progress, and is now being applied to real programs [16,11] well beyond the scale of "toy" examples. These achievements are impressive, but still require massive efforts and highly-trained experts. One of the biggest remaining obstacles is understanding failed verification attempts [19]. Most difficulties in this area stem from inherent limits of static verification, and hence could benefit from complementary *dynamic* techniques.

Static program proving techniques—implemented in tools such as Boogie [17], Dafny [18], and VeriFast [8]—are necessarily incomplete, since they target undecidable problems. Incompleteness implies that program verifiers are "best effort": when they fail, it is no conclusive evidence of error. It may as well be that the specification is sound but insufficient to prove the implementation correct; for example, a loop invariant may be too weak to establish the postcondition. Even leaving the issue of incomplete specifications aside, the feedback provided by failed verification attempts is often of little use to understand the ultimate source of failure. A typical error message states that some executions might violate a certain assertion but, without concrete input values that trigger the violation, it is difficult to understand which parts of the programs

[⋆] Work partially supported by SNF grants 200021-137931 (FullContracts), 200020-134974 (LSAT), and 200021-134976 (ASII); and by ERC grant 291389 (CME).

should be adjusted. And even when verification is successful, it would still be useful to have "sanity checks" in the form of concrete executions, to increase confidence that the written specification is not only consistent but sufficiently detailed to capture the intended program behavior.

Dynamic verification techniques are natural candidates to address these shortcomings of static program proving, since they can provide concrete executions that conclusively show errors and help narrow down probable causes. Traditional dynamic techniques based on *testing* are, however, poor matches to the capabilities of static provers. Testing typically targets simple properties, such as out-of-bound and null dereferencing errors, or, only in a minority of cases, lightweight executable specifications (e.g., contracts). Program provers, in contrast, work with very expressive specification and implementation languages supporting features such as nondeterminism, unbounded quantification, infinitary structures (sets, sequences, etc.), and complex first- or even higher-order axioms; none of these is executable in the traditional sense. As we argue in Sec. 2, however, even relatively simple programs may require such complex specifications. Program provers also support modular verification, where sufficiently detailed specifications of modules or routines are used in lieu of missing or incomplete implementations; this is another scenario where runtime techniques fall short because they require complete implementations.

In this paper, we propose a technique to generate executions of programs annotated with complex specifications using features commonly supported by program provers (nondeterminism, unbounded quantification, partial implementations, etc.). The technique combines symbolic execution with SMT constraint solving to generate small and readable test cases that expose errors (failing executions) or validate specifications (passing executions).

The proposed approach supports executing both imperative and declarative program elements, which accommodates the *implementation* semantics of loops and procedure calls, defined by their bodies, as well as their *specification* semantics, used in modular verification, where the effect of a procedure call is defined solely the procedure's pre- and postcondition and the effect of a loop by its invariant. The implementation semantics is useful to discriminate between inconsistent and incomplete specifications; while the specification semantics makes it possible to generate executions in the presence of partial implementations, as well as to expose spurious executions permitted by incomplete specifications.

Our technique simplifies the constraints passed to the SMT solver, only targeting the values required for a particular symbolic execution. This avoids the solver getting bogged down when reasoning about complex specifications—a problem often arising with program provers—without need for additional guidance in the form of quantifier instantiation heuristics. The simplification also improves the predictability of test case generation. Combined with model minimization techniques, it produces short—often minimal-length—executions that are quite easy to read. While constraint simplification might also produce false positives (infeasible executions), the evaluation of Sec. 5 shows that this rarely happens in practice: the small risk amply pays off by producing easy-to-understand executions, symptomatic of the rough patches in the implementation

```
 1 procedure Max(N: int, a: [int] int) returns (max: int)
 2   ensures (∀ j: int • 0 ≤ j ∧ j < N ⟹ a[j] ≤ max);
 3   ensures (∃ j: int • 0 ≤ j ∧ j < N ∧ a[j] = max);
 4 {
 5   var i: int;
 6   i := 0;
 7   max := 0;
 8   while (i < N) {
 9     if (a[i] > max) { max := a[i]; }
10     i := i + 1;
11   }
12 }
```

Fig. 1. Boogie procedure `Max` that finds the maximum element in an array. Both the specification and the implementation contain errors, and no loop invariant is provided.

or specification that require further attention. We also identify a subset of the annotation language for which no infeasible executions are generated.

We implemented our technique for the Boogie intermediate verification language, used as back-end of numerous program verifiers [18,4,26]. Working atop an intermediate language opens up the possibility of reusing the tool with multiple high-level languages and verifiers that already translate to Boogie. It also ensures that our technique is sufficiently general: Boogie is a small yet very expressive language (including both specification and imperative constructs), designed to support translations of disparate notations with their own supporting methodologies. Our implementation is available as a tool called Boogaloo, distributed as free software: `https://bitbucket.org/nadiapolikarpova/boogaloo/` and accessible through the web: `http://cloudstudio.ethz.ch/comcom/#Boogaloo`. For simplicity, in the paper we will use "Boogaloo" to denote the execution generation technique as well as its implementation, and will employ the self-explanatory Boogie syntax in the examples.

2 Illustrative Example

We give a concise overview of the capabilities of Boogaloo using a simple verification example: finding the maximum element in an integer array.[1] Fig. 1 shows a straightforward Boogie implementation as procedure `Max`, which inputs an integer `N`, denoting the array size, and a map[2] `a` that represents the array elements `a[0]`, ..., `a[N-1]`; it returns an integer `max` for `a`'s maximum. The Boogie procedure includes specification in the form of two postconditions (**`ensures`**), formalizing the definition of maximum: `max` should be no smaller than any element of `a` (line 2); and it should be an element of `a` (line 3).

[1] The tool output messages in this section are abridged without sacrificing the gist of the original.
[2] In general, maps have an infinite domain in Boogie

What happens if we try to verify procedure `Max`, as shown in Fig. 1, using Boogie? Verification fails with a vague error message (“`Postconditions on lines 2 and  3 might not hold.`”) which is inconclusive and of little help to understand the source of failure. Rather, running Boogaloo on the same input generates concrete inputs that make the program fail; we get the message “`Postcondition on line 3 violated with N -> 0, a -> [], max -> 0`”, which clearly singles out a problem with `Max`: the maximum of an empty array is undefined.

We can formalize the expectation that `Max` ought to be a partial function (undefined for empty arrays) as the precondition **`requires`** `N > 0`. Boogie’s output does not, however, change if we add such a precondition: it still cannot establish either postcondition since it would need a loop invariant to reason about loops—no matter how simple they are. Instead, running Boogaloo on `Max` annotated with the precondition shows another input that triggers a failure: “`Postcondition on line 3 violated with N -> 1, a -> [0 -> -1], max -> 0`”. This time the problem is with the implementation rather than with the specification: when `a` contains a single negative value, initializing `max` to `0` (line 7) does not work. With this concise concrete counterexample, it is easy to understand that the same problem occurs with any array containing only negative elements. Designing a correction is also routine: we change the initialization on line 7 to `max := a[0]`, which is well-defined thanks to the precondition `N > 0`.

We can see that the modified program—including precondition and new initialization of `max`—is finally correct. However, Boogie’s behavior on it does not change at all: without a loop invariant, it still fails to prove either postcondition. Boogaloo, in contrast, can generate a number of test cases (1024 by default, which takes just a few seconds with the running example on standard desktop hardware) and successfully check all of them against the specification. While this still falls short of a formal correctness proof, it provides evidence that the program is indeed correct, and that all we have to do is strengthen the specification by adding a suitable loop invariant.

While we selected a simple example which can be briefly presented, we were able to demonstrate, in a nutshell, several fundamental issues of working with static program verifiers such as Boogie, and how Boogaloo can complement their weaknesses. Specifically, Boogaloo’s capabilities to provide concrete inputs that show errors or amass evidence for correctness; and to work with the same programs used for verification including elements such as first-order quantification (lines 2 and 3), but without requiring specifications at all costs (a loop invariant). Another distinguishing, and practically crucially important, feature of Boogaloo is that it produces small (often minimal) tests: in the example, the smallest arrays and the smallest integer values exposing faults and discrepancies.

Comparison with other approaches. To further demonstrate the unique features of Boogaloo, let us consider the behavior of other approaches to complementing static program verification on the same example of procedure `Max`.

Assuming `Max` were a Boogie encoding produced from some high-level programming language, we could use standard testing tools on the source program to generate concrete inputs and discover failures. One problem is that first-order quantifications (and other features used by Boogie) are inexpressible using the simple Boolean expressions of standard programming languages. While the quantifications used in `Max` are bounded, and hence expressible using executable constructs such as finite iterations over arrays or list comprehensions, getting rid of quantifiers and other non-executable constructs is neither possible nor desirable in general. As soon as we look at examples more complex than `Max`, we need to express abstract properties potentially involving infinitely many elements, such as for framing and for reasoning about unbounded sequences of pointers to heap-allocated data. Even in an example as simple as sorting, if a sorting procedure takes a function pointer as argument to denote the comparison function, we need to express that it encodes a total order—something involving quantification over a potentially unbounded domain. More generally, we designed Boogaloo to work with the same proof-oriented annotated programs used by static verifiers, which involve features difficult to execute and normally not found in high-level programming languages.

Another option to debug `Max` is using the Boogie Verification Debugger (BVD [14]), which extracts concrete counterexamples from failed verification attempts. The relevance of such counterexamples is, however, limited in the presence of loops and procedure calls with incomplete specifications. On `Max` as in Fig. 1, BVD returns the assignment "`N = 1, a = [], max = -900`"; after adding `N > 0` as precondition, it returns "`N = 797, a = [], max = -900`"; after fixing the implementation, it returns "`N = 797, a = [0 -> -901], max = -901`". These examples fail to point out the two errors in `Max`, because according to modular reasoning [17] and in the absence of an invariant, any loop is equivalent to assigning arbitrary values to program variables. While BVD's modular semantics helps debug incompleteness in specifications, it also enforces an "all-or-nothing" development style, where developers first have to get right the most complicated part (the invariants), before they can proceed with debugging the rest of the program. This lack of incrementality is what makes modular verification so hard in the first place.

It is possible to make Boogie use loop and procedure bodies instead of their specification by *unrolling* loops and *inlining* procedures U times, for a given $U \geq 0$. With unrolling, BVD finds counterexamples for executions where `N` $\leq U$, and in particular the same counterexamples constructed by Boogaloo. The approach, however, has its limitations. First, unrolling and inlining require users to guess a suitable U; since all longer executions are ignored, verification vacuously succeeds when the shortest counterexample requires $> U$ iterations or nested calls, without providing any concrete feedback. Second, unrolling of complex loops and inlining of recursive procedures scale poorly, as they consist of literally rewriting the code U times; Boogaloo, in contrast, uses symbolic execution techniques, which are less likely to incur blow up. Building a debugger on top of the Boogie verifier also means that it cannot generate passing executions (Boogie does not produce a model in case verification succeeds) and cannot help when the theorem prover gets bogged down. In contrast, Boogaloo uses simpler verification conditions, designed for predictable generation and readability of counter examples as opposed to sound proofs.

3 A Runtime Semantics of Boogie Programs

This section describes the syntax of Boogaloo programs (Sec. 3.1) and their operational semantics (Sec. 3.2). We use the following notation: $\mathbb{Z}$ is the set of mathematical integers; and $\mathbb{B}$ is the set $\{\top, \bot\}$ of Boolean values. A *map* m is a mathematical function from a domain $D_1 \times \cdots \times D_n$, for $n \geq 0$, to a codomain D_0; square brackets denote map applications. Whenever convenient, we see m as a set of $(n+1)$-tuples: $m \subset D_1 \times \cdots \times D_n \times D_0$ such that $(d_1, \ldots, d_n, d_0) \in m$ iff $m[d_1, \ldots, d_n] = d_0$. $\mathrm{dom}(m)$ and $\mathrm{rng}(m)$ denote the *domain* and *range* of m; m is *total* if $\mathrm{dom}(m) = D_1 \times \cdots \times D_n$, and *finite* if $|\mathrm{dom}(m)| \in \mathbb{Z}$; $m[d_1, \ldots, d_n \mapsto d]$ denotes a map m' identical to m except that $m'[d_1, \ldots, d_n] = d$. We overload this notation to denote variable substitution: if $e, y_1, \ldots, y_n$ are expressions, and $x_1, \ldots, x_n$, are variable names, $e[x_1, \ldots, x_n \mapsto y_1, \ldots y_n]$ denotes e with all occurrences of x_k replaced by y_k, for $k = 1, \ldots, n$.

3.1 Input Language

Boogaloo desugars generic Boogie programs [17] into the simpler language described in Fig. 2. Programs P are lists of declarations D, whose order is immaterial. Declarations include uninterpreted **type**s, global **var**iables, and **procedure**s with input parameters, output parameters (**returns**), global variables the procedure may modify (**modifies** clause), and body (between braces). Procedure bodies consist of local variable declarations followed by a list of labeled statements S. The latter include sequential composition, regular and nondeterministic assignment (:= and **havoc**, possibly in parallel to multiple variables), procedure **call**, **assume**, nondeterministic **goto** a set of label identifiers, and abrupt **return** to the caller procedure, as well as *directives* R described shortly. Expressions must be properly typed as **bool**eans, **int**egers, maps $[T_1, \ldots, T_n]T_0$ from arbitrary domain $(T_1, \ldots, T_n)$ and codomain T_0 types, and user-defined uninterpreted types[3]. Expressions E include literal constants C, variables V, map applications $m[t_1, \ldots, t_n]$, map updates $m[t_1, \ldots, t_n := t]$, **old** expressions which refer to the value of an expression when the procedure was entered, plus the usual applications of unary operators UOp, binary operators BOp, a ternary **if**/**then**/**else** operator, and quantifications and lambda expressions QOp.

The *directives* **halt**, **abort**, and **pick** are Boogaloo-specific and characterize symbolic executions: **halt** terminates the current execution with success (marking *passing* executions); **abort** also terminates the current execution but with error (marking *failing* executions); **pick** forces the interpreter to resolve nondeterminism by trying to build a concrete state out of the current symbolic constraints. Boogaloo automatically inserts a **halt** at the end of every control path in the input program; and uses **abort** to desugar **assert** statements as follows. A Boogie statement **assert** `B`, where `B` is a Boolean expression, indicates that `B` must hold in every non-failing execution reaching the statement; **assume** `B`, on the other hand, indicates that only executions where `B` holds upon reaching the **assume** are feasible. Therefore, Boogaloo

[3] While Boogaloo supports Boogie's type constructors with arguments, as well type parameters in procedures and maps, we do not include them in the discussion for simplicity.

$$
\begin{array}{rcl}
P & ::= & D^* \\
D & ::= & \texttt{type}\ tid \mid \texttt{var}\ V : T \\
 & \mid & \texttt{procedure}\ pid\ (\ \langle V : T\rangle^*\ )\ \texttt{returns}\ (\ \langle V : T\rangle^*\ )\ \texttt{modifies}\ (V^*)\ \langle \texttt{\{}\ \langle V : T\rangle^*\ \langle lid : S\rangle^*\ \texttt{\}}\rangle^? \\
S & ::= & S\texttt{;}S \mid \texttt{havoc}\ V^+ \mid V^+ := E^+ \mid \texttt{call}\ V^* := pid\ (E^+) \mid \texttt{assume}\ E \mid \texttt{goto}\ lid^+ \mid \texttt{return} \mid R \\
R & ::= & \texttt{halt} \mid \texttt{abort} \mid \texttt{pick} \\
T & ::= & \texttt{bool} \mid \texttt{int} \mid [\,T^+\,]\,T \mid tid \\
E & ::= & C \mid V \mid E\,[\,E^+\,] \mid E\,[\,E^+ := E\,] \mid \texttt{old}\ E \\
 & \mid & UOp\ E \mid E\ BOp\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \mid QOp\ \langle V : T\rangle^+ \bullet E \\
V & ::= & vid \qquad C\ ::=\ \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \cdots \\
UOp & ::= & - \mid \neg \qquad BOp\ ::=\ + \mid - \mid \cdots \mid < \mid \leq \mid = \mid \cdots \mid \wedge \mid \vee \mid \cdots \qquad QOp\ ::=\ \exists \mid \forall \mid \lambda
\end{array}
$$

Fig. 2. Desugared language supported by Boogaloo, consisting of programs P, declarations D, statements S, types T, and expressions E. Angular brackets $\langle\ \rangle$ are part of the grammar metalanguage, used to mark optional (?) or repeated ($*$, $+$) expressions.

expresses the semantics of **`assert`** `B` using **`assume`**, **`abort`**, and nondeterministic choice as: **`goto`** `T, F;` `F:` **`assume`** `¬ B;` **`abort`**`;` `T:` **`assume`** `B`. Boogaloo also injects a **`pick`** statement right before every **`halt`** and **`abort`**, so that every terminating execution gets a concrete state. Boogaloo automatically instruments programs with the directives R, based on different strategies (see Section 5) so that one can use Boogie programs without additional annotations.

The rest of the desugaring of Boogie into the language of Fig. 2 is fairly standard. We rewrite function declarations **`function`** `f`$(T_1, \ldots, T_n)$ **`returns`**(T_0) into constants **`const`** `f`: $[T_1, \ldots, T_n]\,T_0$ of map type, and express the corresponding function definitions as axioms. In turn, we express axioms and other specification constructs—**`where`** clauses (used to constrain the values of uninitialized variables), pre- and postconditions, and loop invariants—using **`assume`** and **`assert`** reflecting the standard semantics. We replace constants with variables. Finally, we transform procedure bodies into sets of *basic blocks* (labeled sequential blocks of code that end with a **`return`** or **`goto`**) using standard techniques [17].

3.2 Runtime Operational Semantics

We now describe an operational semantics for the language in Fig. 2. The presentation focuses on the most interesting aspects while omitting standard details.

Let us start with an informal overview of the basic concepts. The operational semantics describes the effect, on the symbolic state, of executing statements. The symbolic state associates symbolic values to program variables in scope. Executing some statements may involve enforcing constraints between symbolic values; the most obvious example is that of **`assume`** `P`: the symbolic values associated to variables mentioned in `P` must satisfy `P` in every computation that continues after the statement. Therefore, the symbolic state includes *constraints* which are updated as execution progresses. Finally, **`pick`** directives select *concrete* values that satisfy the current constraints; executions continue after **`pick`** with the selected concrete state components replacing the corresponding symbolic state components (but subsequent statements will be executed symbolically until the next `pick`). In this sense, symbolic executions are *speculative*, in that the constraints may not have a solution (infeasible executions), and *nondeterministic*, in that the constraints may have more than one solution; `pick` forces the interpreter to resolve nondeterministic choice before continuing. Another source of nondeterminism comes from executing **goto**s with multiple labels; such choices are resolved immediately, resulting in explicit path enumeration. Since Boogaloo injects `pick` statements at every terminating location, it can provide concrete input and output values for every feasible execution, while still availing of symbolic representation to limit the combinatorial explosion introduced by the inherently nondeterministic nature of specifications.

The main source of complexity in executing Boogie programs lies in solving constraints, in particular when they involve universal quantifiers and uninterpreted maps with infinite domains. Even though state-of-the-art SMT solvers can decide satisfiability of quantified formulas in many practical cases, they can hardly generate readable "natural" infinite models. In light of these difficulties, we drop Boogie's standard interpretation—where all maps are total—and replace it with a *finitary* interpretation where maps have finite domains. Finite, small instances are sufficient to expose errors and inconsistencies in most programs; Alloy's techniques are based on a similar "small scope" hypothesis [7]. We also treat universally quantified constraints in a special way: the `pick` directive *finitizes* them, that is turns them into simpler quantifier-free constraints. Finitization is in general unsound, but Sec. 5 demonstrates that the precision loss is normally acceptable, especially if the goal is finding inconsistencies and errors.

Concrete values. Each Boogie type corresponds to a set of concrete values: **bool** corresponds to $\mathbb{B}$, **int** corresponds to $\mathbb{Z}$; each user-defined **type** U corresponds to a countable uninterpreted set U; each map type $[\mathsf{T}_1, \ldots, \mathsf{T}_n]\,\mathsf{T}_0$ corresponds to the set of all *finite* maps from $T_1 \times \cdots \times T_n$ to T_0, where T_k is the set of concrete values of type T_k, for $k = 0, \ldots, n$. K denotes the union of all concrete value sets.

Symbolic values correspond to the set Σ defined as:

$$\Sigma \ ::= \ K \mid L \mid \mathit{UOp}\ \Sigma \mid \Sigma\ \mathit{BOp}\ \Sigma \mid \mathbf{if}\ \Sigma\ \mathbf{then}\ \Sigma\ \mathbf{else}\ \Sigma\,,$$

where K is the set of concrete values defined above; unary UOp and binary BOp operators are in Fig. 2, and L denotes a set of *logical variables* of the same types as the concrete values. A logical variable ℓ of type T corresponds to a symbolic placeholder (a "promise") for a concrete value of type T. To represent quantifiers in constraints, we also introduce a set of *universal symbolic values* $\Sigma_\forall ::= \forall \langle V : T \rangle^+ \bullet \Sigma_V$, where Σ_V is a symbolic expression, which can also include bound variables V. Given a set X of expressions, $\mathrm{LV}(X)$ is the set of all logical variables appearing in X.

Symbolic states. A symbolic state (environment) is a tuple $\mathcal{E} = \langle \sigma, \lambda, \mu, \kappa, \upsilon, \tau \rangle$, where the store $\sigma : V \to \Sigma$ maps variables to symbolic values; the logical store $\lambda : L \to \Sigma$ maps scalar logical variables to symbolic values; the map store $\mu : L \to (\Sigma^* \to \Sigma)$ maps map logical variables to symbolic maps; $\kappa \subset \Sigma$ is a finite set of *simple state constraints*; $\upsilon \subset \Sigma_\forall$ is a finite set of *universal state constraints*; and τ is one of ✧, ✓, ✗, denoting an intermediate state (✧), or the final state of a passing (✓) or failing (✗) execution. The map store associates logical variables of map type to *symbolic maps*: finite maps whose domain and range are in Σ; symbolic maps extend

$$\text{LOG-IN}\ \frac{\ell \in \mathrm{dom}(\lambda)}{[\![\ell]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\ \lambda[\ell]} \qquad \text{LOG-OUT}\ \frac{\ell \notin \mathrm{dom}(\lambda)}{[\![\ell]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\ \ell}$$

$$\text{VAR-IN}\ \frac{v \in \mathrm{dom}(\sigma) \qquad [\![\sigma[v]]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\ e}{[\![v]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\ e} \qquad \text{VAR-OUT}\ \frac{v \notin \mathrm{dom}(\sigma) \qquad \ell \text{ is fresh} \qquad \sigma' = \sigma[v \mapsto \ell]}{[\![v]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \ell}$$

$$\text{SEL-IN}\ \frac{[\![(m, \boldsymbol{a})]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ (\ell_m, \boldsymbol{a}') \qquad \boldsymbol{a}' \in \mathrm{dom}(\mu'[\ell_m]) \qquad [\![\mu'[\ell_m][\boldsymbol{a}']]\!]\ {}^{\mathcal{E}'}{=}^{\mathcal{E}'}\ e}{[\![m[\boldsymbol{a}]]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ e}$$

$$\text{SEL-OUT}\ \frac{\ell \text{ is fresh} \qquad [\![(m, \boldsymbol{a})]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1}\ (\ell_m, \boldsymbol{a}_1) \qquad \boldsymbol{a}_1 \notin \mathrm{dom}(\mu_1[\ell_m]) \qquad m' = [\mu_1[\ell_m][\boldsymbol{a}_1] \mapsto \ell]}{[\![m[\boldsymbol{a}]]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \ell \qquad \mathcal{E}' = \langle \sigma_1, \mu_1[\ell_m \mapsto m'], \upsilon_1 \rangle}$$

$$\text{UPD}\ \frac{\ell \text{ is fresh} \qquad [\![(m, \boldsymbol{a}, e)]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1}\ (\ell_m, \boldsymbol{a}_1, e_1) \qquad m' = [\mu_1[\ell_m][\boldsymbol{a}_1] \mapsto e_1]}{[\![m[\boldsymbol{a} := e]]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \ell \qquad \mathcal{E}' = \langle \sigma_1, \mu_1[\ell \mapsto m'], \upsilon_1 \cup \{\forall \boldsymbol{x} \bullet \boldsymbol{x} \neq \boldsymbol{a}_1 \Rightarrow \ell[\boldsymbol{x}] = \ell_m[\boldsymbol{x}]\}\rangle}$$

$$\text{LAMBDA}\ \frac{\ell \text{ is fresh} \qquad [\![e]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1}\ e_1 \qquad \sigma_1(\boldsymbol{x}) = \boldsymbol{\ell}_1}{[\![\lambda \boldsymbol{x} \bullet e]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \ell \qquad \mathcal{E}' = \langle \sigma_1, \mu_1, \upsilon_1 \cup \{\forall \boldsymbol{x} \bullet e_1[\boldsymbol{\ell}_1 \mapsto \boldsymbol{x}]\}\rangle}$$

$$\text{QUANT-T}\ \frac{\mathsf{Skolem}[Q_1 x_1 \cdots Q_n x_n \bullet q]\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1}\ \forall \boldsymbol{y} \bullet q_1 \qquad [\![q_1]\!]\ {}^{\mathcal{E}_1}{=}^{\mathcal{E}_2}\ q_2 \qquad \sigma_2(\boldsymbol{y}) = \boldsymbol{\ell}}{[\![Q_1 x_1 \cdots Q_n x_n \bullet q]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \top \qquad \mathcal{E}' = \langle \sigma_2, \mu_2, \upsilon_2 \cup \{\forall \boldsymbol{y} \bullet q_2[\boldsymbol{\ell} \mapsto \boldsymbol{y}]\}\rangle} \qquad \text{QUANT-F}\ \frac{[\![\widetilde{Q}_1 x_1 \cdots \widetilde{Q}_n x_n \bullet\ \neg q]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \top}{[\![Q_1 x_1 \cdots Q_n x_n \bullet q]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \bot}$$

Fig. 3. Symbolic evaluation (significant cases).

their finite domains as execution progresses; `pick` concretizes their domain and range, turning symbolic maps into concrete ones.

Expression evaluation. Let E denote the set of all expressions defined by E in Fig. 2 but whose atoms range over $C \cup V \cup L$ (i.e., including logical variables L). The *evaluation* of an expression $e \in \mathsf{E}$ in an environment $\mathcal{E}$ is a symbolic value in Σ. We use the notation: $[\![e]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ e'$ to denote that $e \in \mathsf{E}$ evaluates in $\mathcal{E}$ to $e' \in \Sigma$. As we detail shortly, evaluating an expression may change the environment; correspondingly, $\mathcal{E}'$ denotes the updated environment, whose components are written $\langle \sigma', \lambda', \mu', \kappa', \upsilon', \tau' \rangle$. When convenient, we extend this notation to *sequences* $\boldsymbol{e} = e_1, \ldots, e_n$ of expressions, evaluated one after another as follows: $[\![\boldsymbol{e}]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \boldsymbol{e}'$ iff $[\![e_k]\!]\ {}^{\mathcal{E}_k}{=}^{\mathcal{E}'_k}\ e'_k$ for $k = 1, \ldots, n$, $\mathcal{E}'_k = \mathcal{E}_{k+1}$ for $k = 1, \ldots, n-1$, and $\boldsymbol{e}' = e'_1, \ldots, e'_n$. Fig. 3 shows the evaluation rules for the most interesting expression kinds. Since evaluation does not change the λ, κ, and τ environment components, Fig. 3 omits them. Also notice that evaluating a symbolic value never changes the environment, and every concrete value evaluates to itself.

Rules LOG-IN and LOG-OUT describe the simple cases of evaluating a logical variable ℓ: if $\lambda[\ell]$ is defined, it yields ℓ's evaluation; otherwise, ℓ evaluates to itself. Since λ is only changed by executing `pick`, which assigns concrete values to all logical variables in κ, λ evaluates to concrete values when defined: $\mathrm{LV}(\mathrm{rng}(\lambda)) = \emptyset$.

Rules VAR-IN and VAR-OUT describe the evaluation of a (program) variable v. If it has already been initialized, the evaluation of $\sigma[v]$ gives its symbolic value. Otherwise (VAR-OUT), such as when v enters the scope or after executing **havoc** v, $\sigma[v]$ gets initialized to a fresh logical variable ℓ.

The rules for map selection are similar to those for variables but target the map store μ: if a map selection has already been evaluated, its symbolic value is returned (SEL-IN); otherwise, a fresh logical variable is generated and stored in μ (SEL-OUT).

Rules UPD and LAMBDA deal with evaluating expressions of map type for updates and lambda abstractions. Both rules introduce a fresh map logical variable and add to υ a universally quantified constraint that defines the map. Thus, map expressions (variables, updates, and lambdas) always evaluate to a logical variable; this justifies using the evaluation ℓ_m of m as an index in μ in the premises of SEL-IN, SEL-OUT, and UPD.

The rules for quantified expressions are non-deterministic. Consider an expression $\mathcal{Q} = Q_1 x_1 \cdots Q_n x_n \bullet q$ in prenex normal form, where $n > 0$, Q_k is one of $\forall$ and $\exists$ for all k's, and q is quantifier-free. Rule QUANT-T evaluates $\mathcal{Q}$ to true and adds it to the universal constraints υ after the following transformation. First, $\mathcal{Q}$ is Skolemized as $\forall \boldsymbol{y} \bullet q_1$, where $\boldsymbol{y}$ is the subsequence of $x_1, \ldots, x_n$ including only those x_k's for which Q_k is $\forall$; $\mathcal{E}_1$ is the environment after Skolemization, which contains fresh logical variables for the Skolem functions introduced by the process. Evaluating q_1 in $\mathcal{E}_1$ yields some q_2 where the bound variables $\boldsymbol{y}$ map to fresh logical variables $\boldsymbol{\ell}$; after performing the substitution $q' = q_2[\boldsymbol{\ell} \mapsto \boldsymbol{y}]$, $\forall \boldsymbol{y} \bullet q'$ is added to υ. In the special case where $\mathcal{Q}$ is purely existential, Skolemization yields a quantifier-free formula, and the corresponding q_2 is directly added to κ. Rule QUANT-F, which evaluates $\mathcal{Q}$ to false, follows by duality ($\widetilde{\forall}$ denotes $\exists$, and $\widetilde{\exists}$ denotes $\forall$).

Procedure call semantics. The precise semantics of procedure calls involves several details to support recursion—mainly, maintaining a stack of environments and correspondingly keeping track of scope. We overlook these tedious aspects and focus on the gist of the semantics of a call to a generic procedure P (before desugaring):

$$\texttt{\textbf{procedure} P (}\boldsymbol{a}\texttt{) \textbf{returns} (}\boldsymbol{o}\texttt{) \textbf{requires} } p \texttt{ \textbf{ensures} } q \texttt{ \textbf{modifies}(}\boldsymbol{m}\texttt{)} \; \langle\{B\}\rangle^?$$

with formal input $\boldsymbol{a}$ and output $\boldsymbol{o}$ parameters, modified global variables $\boldsymbol{m}$, body B, and pre- and postcondition p and q. The desugaring of Sec. 3.1 turns pre- and postcondition into checks at the call site:

$$\texttt{\textbf{assert} } p[\boldsymbol{a} \mapsto \boldsymbol{u}]\texttt{; \textbf{call} } \boldsymbol{v} := \texttt{P}(\boldsymbol{u})\texttt{; \textbf{assume} } q[\boldsymbol{a}, \boldsymbol{o} \mapsto \boldsymbol{u}, \boldsymbol{v}]\texttt{;}$$

(For brevity, we do not discuss the handling of **old** expressions in postconditions.) It also generates a modified procedure body B' to reflect the implementation or specification semantics, according to whether P has a body or not: if B is defined, B' adds an **assert** q before each **return** statement in B; if B is not defined, B' consists of the single statement **havoc** $\boldsymbol{o}, \boldsymbol{m}$ (which, combined with assuming the postcondition at the call site, renders the specification semantics). The effect of the call statement is then given by $B'[\boldsymbol{a} \mapsto \boldsymbol{u}]$@entry where $B'[\boldsymbol{a} \mapsto \boldsymbol{u}]$ is a shorthand for B' with actual arguments replaced for formals and @entry denotes the basic block of statements at procedure P's entry. Even though Boogaloo defaults to implementation semantics whenever a body is available, one can always switch to the specification semantics for a particular procedure by commenting out its body.

Operational semantics. Fig. 4 describes the operational semantics of statements other than procedure calls, using the notation $\mathcal{E} \xrightarrow{\;\texttt{S}\;} \mathcal{E}'$ to denote that executing state-

$$\text{SEQ}\,\frac{\tau = \diamondsuit \quad \mathcal{E} \;-\mathtt{I}\!\rightsquigarrow \mathcal{E}_1 \quad \mathcal{E}_1 \;-\mathtt{J}\!\rightsquigarrow \mathcal{E}'}{\mathcal{E} \;-\mathtt{I};\mathtt{J}\!\rightsquigarrow \mathcal{E}'} \qquad \text{GOTO}\,\frac{\tau = \diamondsuit \quad k \in \{1, \dots, n\} \quad \mathcal{E} \;-@\mathsf{x}_k\!\rightsquigarrow \mathcal{E}'}{\mathcal{E} \;-\mathbf{goto}\ \mathsf{x}_1, \dots, \mathsf{x}_n\!\rightsquigarrow \mathcal{E}'}$$

$$\text{RETURN}\,\frac{\tau = \diamondsuit \quad \mathcal{E} \;-@\mathsf{caller}\!\rightsquigarrow \mathcal{E}'}{\mathcal{E} \;-\mathbf{return}\!\rightsquigarrow \mathcal{E}'} \qquad \text{ASSUME}\,\frac{\tau = \diamondsuit \quad [\![\mathsf{P}]\!]\ {}^{\mathcal{E}}\!\!=^{\mathcal{E}'} p}{\mathcal{E} \;-\mathbf{assume}\ \mathsf{P}\!\rightsquigarrow \langle \sigma', \lambda', \mu', \kappa' \cup \{p\}, \upsilon', \tau' \rangle}$$

$$\text{HAVOC}\,\frac{\tau = \diamondsuit}{\mathcal{E} \;-\mathbf{havoc}\ \mathsf{v}\!\rightsquigarrow \langle \sigma \setminus \{(\mathsf{v}, \sigma[\mathsf{v}])\}, \lambda, \mu, \kappa, \upsilon, \tau \rangle} \qquad \text{ASSIGN}\,\frac{\tau = \diamondsuit \quad [\![\mathsf{e}]\!]\ {}^{\mathcal{E}}\!\!=^{\mathcal{E}'} e'}{\mathcal{E} \;-\mathsf{v} := \mathsf{e}\!\rightsquigarrow \langle \sigma[\mathsf{v} \mapsto e'], \lambda', \mu', \kappa', \upsilon', \tau' \rangle}$$

$$\text{HALT}\,\frac{\tau = \diamondsuit}{\mathcal{E} \;-\mathbf{halt}\!\rightsquigarrow \langle \sigma, \lambda, \mu, \kappa, \upsilon, ✓ \rangle} \qquad \text{ABORT}\,\frac{\tau = \diamondsuit}{\mathcal{E} \;-\mathbf{abort}\!\rightsquigarrow \langle \sigma, \lambda, \mu, \kappa, \upsilon, ✗ \rangle}$$

$$\text{PICK}\,\frac{\tau = \diamondsuit \quad \kappa' = \kappa \cup \kappa_\mu \cup \Phi(\upsilon) \quad \mathrm{dom}(\Lambda) = \{\ell \in \mathrm{LV}(\kappa') \mid \ell \text{ is scalar}\} \quad \mathcal{E}' = \langle \sigma, \lambda \cup \Lambda, \mu, \emptyset, \upsilon, \tau \rangle \quad [\![\bigwedge \kappa']\!]\ {}^{\mathcal{E}'}\!\!=^{\mathcal{E}'} \top}{\mathcal{E} \;-\mathbf{pick}\!\rightsquigarrow \mathcal{E}'}$$

Fig. 4. Symbolic execution: operational semantics. All rules describe transformations of a generic symbolic state $\mathcal{E} = \langle \sigma, \lambda, \mu, \kappa, \upsilon, \tau \rangle$.

ment S changes the environment $\mathcal{E}$ into $\mathcal{E}'$. Rules are applicable only if $\tau = \diamondsuit$, that is if the computation has not terminated yet; after rules HALT or ABORT have changed τ to passing ✓ or failing ✗ no rule is applicable and hence the computation terminates.

Rules SEQ for sequential composition, GOTO for branch, and RETURN for abrupt termination are standard, using the notation @caller to denote the basic block beginning after the current call in the caller procedure. Rule GOTO is clearly nondeterministic.

Rule ASSUME adds the assumed Boolean formula to the set κ of state constraints. Rule HAVOC "forgets" the symbolic value of the variable v, as if uninitialized. Rule ASSIGN updates the symbolic value in σ associated to the assigned variable v.

The most interesting rule is PICK, which details how **pick** concretizes symbolic states. It extends κ into κ', adding map instance constraints $\kappa_\mu = \{m[\boldsymbol{x}] = y \mid (m, \boldsymbol{x}, y) \in \mu\}$, which express the information in μ about symbolic maps; as well as *finitized* universal constraints $\Phi(\upsilon)$. It then picks a *solution* $\Lambda : L \to K$ of κ': an assignment of concrete values to scalar logical variables for which the conjunction of constraints in κ' evaluates to true. It finally adds the picked solution to λ and drops the solved constraints. The rule is nondeterministic, as κ' might have multiple solutions. When κ' has no solutions, the rule cannot apply and executions gets stuck at **pick**: we call such executions *infeasible*. The rule is also agnostic with respect to the exact method of solving simple constraints, as well as to the finitization mapping Φ. To solve constraints one can leverage an external constraint solver or even brute force enumeration; even if the solving process is unsound, one can always evaluate κ' in $\mathcal{E}'$ and discard solutions that do not simplify to $\top$. The only requirement on $\Phi : \Sigma_\forall{}^* \to \Sigma$ is that it is an over-approximation: any valid solution of υ is also a solution of $\Phi(\upsilon)$. In practice, Φ performs quantifier instantiation: it replaces a quantified formula $\forall \boldsymbol{x} \bullet q$ with a finite set of quantifier-free formulas $\{q[\boldsymbol{x} \mapsto \boldsymbol{e}] \mid e \in R\}$, for some finite set R of "relevant" symbolic values. The challenge is to choose an R that is large enough to describe all relevant values in the current environment, yet small enough to produce constraints that can be solved efficiently. Sec. 4 gives more details about Boogaloo's finitization procedure.

Boogaloo vs. Boogie semantics. How does the operational semantics discussed in this section compare with the original Boogie semantics? For this discussion, a *semantics* of a program P is a set of sequences of concrete states, corresponding to its feasible terminating executions; a (concrete) state $\mathcal{C} = \langle \sigma, \tau \rangle$ consists of a store σ (involving finitely many variables) and a termination flag $\tau \in \{✧, ✓, ✗\}$. A state $\mathcal{C}$ is *finitary* if it involves only finite maps: for all $m \in \mathrm{dom}(\sigma)$, $|\mathrm{dom}(\sigma[m])|$ is finite; otherwise, it is *infinitary*. A state $\mathcal{C}_F$ *finitizes* another state $\mathcal{C}$ (written $\mathcal{C}_F \sqsubseteq_{\mathcal{F}} \mathcal{C}$) iff $\mathcal{C}_F$ is finitary, $\tau_F = \tau$, $\mathrm{dom}(\sigma_F) = \mathrm{dom}(\sigma)$ and, for all map variables $m \in \mathrm{dom}(\sigma)$, $\sigma_F[m] \subseteq \sigma[m]$. A sequence e of states is finitary (infinitary) if all its elements are finitary (infinitary); e finitizes another sequence e' if every state of e finitizes the corresponding state of e'.

For a program P, $\mathcal{I}[P]$ denotes the Boogie semantics defined in [17], which is infinitary since all maps are total ($\mathcal{I}$ is for "infinitary"); and $\mathcal{F}[P]$ denotes the finitary semantics of this paper, where all maps have finite domains.[4] Assuming perfect constraint-solving capabilities, the only aspect where $\mathcal{F}$ may drop information w.r.t. $\mathcal{I}$ is in the rule PICK, and more precisely in the finitization mapping Φ. The requirement that Φ be an over-approximation implies that every Boogie execution is finitized by *some* Boogaloo execution. The converse does not hold in general: in particular, for some programs S, $\mathcal{I}[S] = \emptyset$ but $\mathcal{F}[S] \neq \emptyset$ contains executions (which we regard as spurious). For example, the following program:

```
var a: [int] int;
assume (∀ i, j: int • i < j ⟹ a[i] < a[j]);
assume a[0] = 0 ∧ a[1000] = 1;
```

has no feasible executions in $\mathcal{I}$, while the current implementation of Boogaloo produces a passing execution where the quantified constraint is only instantiated for $\mathtt{i} = 0$ and $\mathtt{j} = 1000$. Sec. 5 demonstrates that such unsound executions are infrequent in practice, and, even when they occur, workarounds are possible, for example forcing the evaluation on more points by accessing them in a loop. Also, Boogaloo's implementation of Φ does not produces spurious executions for programs where all quantified constraints are derived from terminating recursive function definition (see Sec. 4).

There is an additional source of discrepancies between $\mathcal{I}$ and $\mathcal{F}$, due to the fact that Boogie always uses the specification semantics for loops and procedure calls, while Boogaloo defaults to the implementation semantics, which might contain fewer executions. This discrepancy between the two semantics is a useful feature, which makes it possible to debug programs in presence of incomplete specifications. The specification semantics is still available on demand in Boogaloo: it is sufficient to replace an imperative construct with its specification.

[4] The soundness of desugaring implies that all feasible executions both in the Boogie and in the Boogaloo semantics agree on being passing or failing.

4 Boogaloo: Implementation Details

This section presents some details of the Boogaloo tool—our prototype implementation of the approach described in the previous sections. The tool takes as input a Boogie source file and a procedure name as entry point, and produces a set of feasible executions, characterized by their concrete initial and final states. Boogaloo is implemented in Haskell, and uses the SMT solver Z3 [5] in the back-end.

Finitizing universal constraints. The choice of the finitization mapping Φ plays an important role. Our experiments suggest that the powerful quantifier instantiation strategies available in SMT solvers such as Z3 have some downsides when applied to solve constraints generated by executing Boogie programs, as their performance is unpredictable unless additional user input (in the form of "triggers") is provided. Instead, Boogaloo preprocesses constraints using a simple strategy, based on the observation that universally quantified formulas are typically used to axiomatize uninterpreted maps; since we are only interested in finitely many points (those stored in μ), we just instantiate the bound variables at those points. Quantified constraints that do not contain map applications are simply ignored; the examples in Sec. 5 suggests that this finitization strategy is not too restrictive on typical verification examples.

This is how Boogaloo implements Φ for a formula $\forall \boldsymbol{x} \bullet P(\boldsymbol{x})$. For each term $m[\boldsymbol{y}]$ in P such that $\boldsymbol{y}$ includes some bound variable (i.e., $\boldsymbol{y} \cap \boldsymbol{x} \neq \emptyset$), Boogaloo extracts a parametrized map constraint of the form $(m, Q(\boldsymbol{y}, m[\boldsymbol{y}]))$, where Q is a subformula of P including the term, and $\boldsymbol{y}$ are the parameters free in Q; if $\boldsymbol{x} \not\subseteq \boldsymbol{y}$, then Q is itself quantified. For example, $\forall i \bullet a[i] > i \wedge b[i, 0] = 1$ determines two parametrized constraints: $(a, a[i] > i)$ and $(b, j = 0 \implies b[i, j] = 1)$; whereas $\forall i, j \bullet i < j \implies c[i] < c[j]$ determines: $(a, \forall j \bullet i < j \implies c[i] < c[j])$.

Boogaloo evaluates parametrized constraints for a given map store μ iteratively: pick an element $p = (m, \boldsymbol{e}, s)$ of μ, instantiate all parametrized constraints for m with $\boldsymbol{e}$ and evaluate them, and mark p; repeat until all elements of μ are marked. If a Q in a parametrized constraint (m, Q) contains quantifiers, instantiating m triggers the generation of new parametrized constraints from Q (also to completion until all constraints are quantifier-free).

Since evaluating a parametrized constraint may add new points to μ, termination of the evaluation procedure is not guaranteed in the presence of recursive formulas, which determine constraints $(m, Q(\boldsymbol{y}, m[\boldsymbol{y}]))$ where Q also contains applications of m to elements other than $\boldsymbol{y}$. For example, an axiomatization of the factorial f as $f[0] = 1$ and $\forall i \bullet i > 0 \implies f[i] = i \cdot f[i - 1]$ determines the constraint $q = (f, i > 0 \implies f[i] = i \cdot f[i - 1])$. If $\mu[f] = (\ell_0, \ell_1)$, evaluating q for $i = \ell_0$ introduces a new map application at $\ell_0 - 1$, which then introduces an application at $\ell_0 - 2$, and so on. Boogaloo evaluates such recursive constraints using fair unrolling similarly to [24], based on the notion of guard: a parametrized constraint is *guarded* if has the form $(m, G(\boldsymbol{y}) \implies B(\boldsymbol{y}))$. When Boogaloo's iterative evaluation picks an element $p = (m, \boldsymbol{e}, s)$, it nondeterministically chooses a subset D of all guarded constraints for m and "disables" them in the evaluation determined by p: for a parametrized constraint $q = (m, G \implies B)$, it evaluates the constraint $\neg G$ if $q \in D$, and $G \wedge B$ otherwise. For the "right" selection of D, recursive definitions are disabled, so that they do not add new points to μ and evaluation terminates. In the factorial example, there are two choices for $f[\ell_0]$: disabling or enabling the guarded constraint. Disabling it terminates the finitization process, producing an execution with $\ell_0 = 0$; enabling the guarded constraints produces one iteration (for $f[\ell_0 - 1]$), which in turn recursively leads to the same two choices, and so on. Unlike [24], which works only with function definitions and thus assumes that guards are mutually exclusive and cover all cases, Boogaloo's fair enumeration applies to guards of any form and constraints other than equality; it also provides an option to limit the number of unrollings, because recursive constraints may be not well-founded (a sufficient condition for termination).

Nondeterminism. There are four sources of nondeterminism in Boogaloo semantics: evaluation of quantified expressions (rules QUANT-T and QUANT-F), `goto`s, and `pick`—involving the disabling of guarded constraints in Φ and constraint solving to select a solution Λ. Boogaloo enumerates nondeterministic choices using backtracking monads (e.g. [9]). The command-line interface currently offers depth-first and fair exploration strategies, but the implementation easily accommodate others parametrically.

When executing `goto` statements, the order in which labels are tried may affect progress: if the first chosen label leads back to the same statement, execution gets stuck in a loop. To avoid this situation, Boogaloo keeps track of how often each label was taken along the current execution path, and always tries labels in ascending order of their frequencies (least frequent first). This strategy also has the nice effect of enumerating shorter executions before longer ones in the long run. A similar strategy applies to disabling parametrized constraints.

Since symbolic computation is speculative, it introduces the risk that long computations are constructed only to realize, when solving the symbolic constraints, that they are infeasible. This risk is mitigated by the enumeration technique, which produces short execution first. Moreover, whenever a constraint evaluates to the concrete value $\perp$, the current execution path is immediately aborted. This mitigates the overhead incurred by nondeterministic evaluation of quantified expressions: such expressions are likely to occur inside `assume` statements, thus branches where they evaluate to false are immediately abandoned. Additionally, Boogaloo transparently tests the satisfiability of the current constraints κ at various points during an execution, and proceeds only if the constraints are satisfiable; unlike `pick` which may enumerate multiple solutions, a satisfiability check does not cause additional nondeterminism. One can still explore the trade-off between few expensive symbolic executions and many cheap concrete executions by adding `pick` directives at arbitrary points.

Minimization. In addition to producing short executions first, Boogaloo also uses a minimization technique based on binary search (similar to the one in [12]) in order to favor *small* integers for concrete values. In our experience, this significantly improves readability: for example, a constraint "x is positive and divisible by 5" with minimization produces the most natural solution $x = 5$ as first witness.

5 Experimental Evaluation

We evaluated Boogaloo on a choice of 15 examples from various sources[5]. Tab. 1 lists the programs and some data about them. The bulk of the evaluation targets the *veri-*

[5] Examples are available online at `http://se.inf.ethz.ch/people/polikarpova/boogaloo/`

fication of algorithms of various kinds, listed in the top part of the table. For each of these problems, we constructed a correct version equipped with consistent but generally incomplete specifications, and a buggy one, obtained by injecting implementation or specification errors. We ran Boogaloo on both versions, with the goal of generating executions: passing executions for the correct programs, and failing executions exposing the bug for the buggy programs. The rest of the programs, in the bottom part of Tab. 1, are examples of *declarative* programming, which exercise Boogaloo's constraint solving capabilities to generate outputs satisfying given properties, in the absence of imperative implementations. We now briefly mention the most interesting features of our examples, and summarize the experimental results.

PROGRAM	FEATURES	LOC	FUN	ANNOTATIONS						TIME			BUG		
				A	S	U	R	E	I	N	t_Σ	t_C	N	t_Σ	t_C
ArrayMax	see Sec. 2	33	0	0	0	1	1	2	1	26	4	0	0	0	0
ArraySum	recursive definition	34	1	0	0	1	1	1	2	26	75	2	1	0	0
BinarySearch	complex precondition	49	1	0	0	2	2	3	2	26	2	0	0	0	0
BubbleSort	complex postcond. and invariants	74	1	0	0	2	1	4	5	6	80	21	2	0	0
DutchFlag	user-defined types [6]	96	3	0	0	2	2	8	6	7	132	43	1	0	0
Fibonacci	recursive procedure	40	1	3	1	0	2	0	0	11	88	0	0	0	0
Invert	complex pre- and postconditions	37	0	0	0	3	3	2	1	9	19	118	2	0	2
LinkedListTraversal	heap model	49	3	2	0	0	1	1	1	8	50	54	2	0	0
ListInsert	see [14]	52	1	0	0	2	1	1	0	6	78	5	1	0	0
QuickSort	helper and recursive procedures	89	3	0	0	2	1	6	0	3	4	1	2	∞	0
QuickSort PI	partial implementation	79	3	0	0	2	2	9	0	4	∞	64	2	1	0
TuringFactorial	unstructured control flow	37	1	2	5	0	1	1	0	11	1	0	3	0	0
Split	linear arithmetic [13]	22	0	0	0	0	1	3	0	–	0	0			
SendMoreMoney	fixed-size array constraints [12]	36	1	0	0	15	0	0	0	–	2	2			
Primes	recursive definition [12]	31	2	0	0	0	0	2	0	8	7	4			
NQueens	variable-size array constraints	37	2	1	0	3	0	0	0	11	45	1			

Table 1. Programs tested with Boogaloo: name, main features, lines of code, number of specification functions, annotations (**`axiom`**s A, **`assert`**s S, **`assume`**s U, **`requires`** R, **`ensures`** E, loop **`invariant`**s I), time to generate passing executions, time to generate failing executions for the buggy version. Times in seconds, rounded to the nearest integer: for a given input size N, time t_Σ with fully symbolic execution and t_C with concretization. ∞ denotes a timeout of 180 seconds.

Verification. The majority of the programs in the top part of Tab. 1 are slightly adapted examples from the Boogie project repository[6], verification competitions [10], or previous work [6,14]; they contain features that exercise various aspects of the test-case generation process. Strong preconditions (such as an array being sorted in BinarySearch or being a permutation in Invert) make generating valid executions challenging using standard testing enumeration techniques. Inlining (available in Boogie) scales poorly with the recursive procedure calls of Fibonacci and QuickSort. The specifications of BinarySearch, BubbleSort, QuickSort, and Invert use nested universal quantifiers with bound variables mentioned in different predicates. QuickSort PI (partial implementation) is a variant of QuickSort whose partitioning procedure has a complete pre- and postcondition but no implementation. This may represent an intermediate development step where we want to validate the overall logic of QuickSort before proceeding with implementing the partitioning procedure. Boogaloo simulates array parti-

[6] `http://boogie.codeplex.com/`

tioning based only on its specification—something unachievable with traditional testing techniques. The injected bugs are mostly off-by-one errors and missing preconditions, both of which frequently occur in practice; the bugs in BinarySearch are among those found in textbooks [22].

Declarative programming. The other four examples come from previous work on constraint programming and code synthesis [13,12], and involve linear arithmetic, recursively defined functions, and quantification over variable-sized arrays. Constraints are declared using **`assume`** statements or procedures without implementation; Boogaloo generates program outputs satisfying the constraints.

Experimental results. All problems in Tab. 1 but two include a parameter N that defines the input size (the input array or list for most problems). Column TIME displays the value of N used in the experiments; and the time required to generate a passing execution with different concretization strategies: t_Σ corresponds to fully symbolic executions where the state is concretized only once after terminating; t_C, instead, corresponds to executions where the state is concretized before every jump statement. Column BUGGY displays the same time measures for the buggy programs; for these, the value of N corresponds to the input size exposing the bug found by Boogaloo (which is the smallest possible for all programs). In all experiments we imposed a timeout of 180 seconds, to reflect the expectation to use Boogaloo with good responsiveness.

Concretizing before jumping generally leads to faster executions, even with an order-of-magnitude difference. This strategy may lead to heavy backtracking when the constraints on a given logical variable are imposed incrementally, with one or more concretization points in between, producing potentially lengthy combinatorial enumerations. In most examples this does not happen—constraints are "local"—and hence concretizing does not degrade performance. The performance difference between fully symbolic and concretized executions for the same example is particularly conspicuous in the current prototype implementation, which uses Z3 through a pure API and hence cannot make use of incremental constraint solving (which would be useful to avoid solving the same constraints multiple times during a single symbolic execution). We speculate that incremental solving would greatly reduce the performance difference between the two concretization strategies. Even though the current implementation has a big potential for improving performance, the experimental results are encouraging: in particular, exposing bugs—the primary purpose of Boogaloo—is fast, even in the presence of partial implementations. Understanding the unexpected behavior with QuickSort, where fully symbolic execution times out, requires further investigation.

6 Related Work

Debugging failed verification attempts. While still an incipient research area, a few techniques have recently been proposed to help understand and debug failed attempts of program verifiers. Sec. 2 already mentioned the Boogie Verification Debugger (BVD, [14]); the Spec# debugger [21] implements similar functionalities which construct concrete counterexamples from failed Boogie runs. Two-step verification [27] compares verification with different semantics (based on unrolling and inlining) to attribute verification failures to either inconsistent or incomplete specifications.

The fact that all these approaches are built around the output provided by a program verifier determines their main limitations compared to Boogaloo. As we demonstrated in Sec. 2, when verification fails because of insufficient specification, the counterexamples generated by BVD or similar tools are typically uninformative or even misleading, because they ignore the implementation even when it is correct (e.g., a loop), unless it is comes with an accurate specification (e.g., a loop invariant). Boogaloo supports a more incremental approach, where users can concentrate on fixing major bugs first. Sec. 2 also discussed how inlining and unrolling (available in Boogie and automatically used in two-step verification) ameliorate these problems, but they are also not directly comparable to Boogaloo, since they scale poorly and require to know explicit unrolling bounds. Of course, the finitary semantics implemented by Boogaloo comes with its own shortcomings: if the shortest counterexamples are very long, it may be infeasible to generate them by enumeration, whereas a static verifier's modular reasoning is insensitive to the length of concrete execution paths since it is entirely symbolic; tools such as BVD can directly work on any failed verifier attempts.

Another approach to produce readable counterexamples is restricting the input language (e.g., [24]), trading off expressiveness for decidability. Bounded model-checking techniques (e.g., [3]) also target standard programming languages and the verification of properties that do not include features such as infinite mappings and unbounded quantification. Boogaloo follows a different course: it supports the entire Boogie language as used in practice, which does not restrict expressiveness a priori, but may produce spurious counterexamples.

Testing is the process of executing programs to make them fail. Since it is based on execution, it is typically limited to violations of simple properties that can be efficiently evaluated at runtime and are implicit in the programming language semantics (e.g., null dereferencing). Languages such as Eiffel [23], JML [15], and Jahob [28] incorporate a richer language for annotations that is still executable, so as to extend the applicability of standard testing techniques. Another line of research in testing is the combination with static techniques, with the goal of complementing each other's strengths to search the input state space more efficiently. In [25], we combined testing with program proving at a high level. A different array of techniques combines testing with symbolic execution; see the recent survey [2]. Boogaloo is also based on symbolic execution, but with a different overall goal; as future work, we will leverage other techniques from symbolic execution to improve the enumeration of executions.

Constraint programming supports program definitions based on declarative constraints, describing properties of the solution, rather than on traditional imperative constructs. Logic programming extends functional programming languages [1]; more recent approaches combine declarative constraints with imperative languages [20,12]. All these approaches restrict the expressiveness of the constraint language to have predictable performance and some guarantees about soundness, completeness, or both. As briefly demonstrated in Sec. 5, Boogaloo can also be used as a Boogie-based constraint programming language. Unless we also restrict the language of assertions, we cannot offer strong guarantees about properties of the executions generated by Boogaloo (see the end of Sec. 3.2 for a discussion). However, the usage as a constraint programming language brings much flexibility to Boogaloo as a testing environment for Boogie programs, since users can achieve different trade-offs between modularity and scalability opting for the implementation or the specification semantics.

7 Conclusions and Future Work

We presented a technique and a prototype implementation to execute programs with complex specifications and nondeterministic constructs, written in the Boogie intermediate verification language.

Among the various directions for future work, let us mention: integrating domain-specific decision procedure to reduce spurious counterexamples; improving the performance by solving constraints incrementally and by pruning infeasible branches; more experiments with automatically generated Boogie translations; and a user study to assess the practical usability alongside Boogie.

References

1. S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, 2010.
2. C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.
3. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*, pages 570–574, 2005.
4. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
5. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
6. C. A. Furia, B. Meyer, and S. Velder. Loop invariants: Analysis, classification, and examples. `http://arxiv.org/abs/1211.4470`, 2012.
7. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
8. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008.
9. O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *ICFP*, pages 192–203. ACM, 2005.
10. V. Klebanov et al. The 1st verified software competition: Experience report. In *FM*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011.
11. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.
12. A. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL*, pages 151–164, 2012.
13. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329. ACM, 2010.
14. C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie verification debugger (tool paper). In *SEFM*, volume 7041 of *LNCS*, pages 407–414. Springer, 2011.
15. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

16. D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 806–809, Berlin, Heidelberg, 2009. Springer-Verlag.
17. K. R. M. Leino. This is Boogie 2, 2008. `http://goo.gl/QsH6g`.
18. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
19. K. R. M. Leino and M. Moskal. Usable auto-active verification. In *Usable Verification Workshop*. `http://fm.csl.sri.com/UV10/`, 2010.
20. A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520. ACM, 2011.
21. P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *FM*, volume 6664 of *LNCS*, pages 73–87. Springer, 2011.
22. R. E. Pattis. Textbook errors in binary searching. In *SIGCSE*, pages 190–194. ACM, 1988.
23. N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *ICSE*, pages 257–266. ACM, 2013.
24. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011.
25. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *SEFM*, volume 7041 of *LNCS*. Springer, 2011.
26. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Verifying Eiffel programs with Boogie. In *BOOGIE workshop*, 2011. `http://arxiv.org/abs/1106.4700`.
27. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Program checking with less hassle. In *VSTTE*. To appear, 2013.
28. K. Zee, V. Kuncak, M. Taylor, and M. C. Rinard. Runtime checking for program verification. In *RV*, volume 4839 of *LNCS*, pages 202–213. Springer, 2007.

A New Experimental Results

PROGRAM	TIME			BUG		
	N	t_{Σ}	t_C	N	t_{Σ}	t_C
ArrayMax	46	0.5	0.4	0	0.0	0.0
	26	0.2	0.2			
ArraySum	46	0.3	0.4	1	0.0	0.0
	26	0.1	0.2			
BinarySearch	46	0.1	0.0	0	0.1	0.2
	26	0.1	0.0			
BubbleSort	11	9.3	113.9	2	0.1	0.3
	6	0.6	6.1			
	15	115.1	∞			
DutchFlag	20	3.8	7.6	1	0.0	0.0
	7	0.3	0.6			
Fibonacci	19	93.9	3.7	0	0.0	0.0
	11	0.3	0.1			
	20	∞	6.4			
Invert	10	48.1	1.4	2	0.0	0.1
	9	11.2	1.0			
	20	∞	13.3			
LinkedListTraversal	20	61.8	2.5	2	0.0	0.0
	8	0.5	0.2			
ListInsert	4	2.4	4.1	1	0.0	0.0
	6	41.4	∞			
	7	164.5	∞			
QuickSort	15	8.4	177.9	2	∞	0.1
	3	0.2	0.3			
	20	36.1	∞			
QuickSort PI	4	0.2	42.1	2	0.1	∞
	13	111.6	∞			
TuringFactorial	21	0.2	0.2	3	0.0	0.0
	11	0.1	0.1			
Split	–	0.0	0.0			
SendMoreMoney	–	0.3	0.3			
Primes	8	0.2	0.9			
	10	∞	5.6			
NQueens	15	1.2	31.8			
	11	0.2	0.8			
	25	26.6	∞			

Table 2. New experimental results with Boogaloo v. 0.4.1/2013-09-18 on the same programs as in Table 1. For each program, the table lists the time performance: on the largest instance size (N) among those tried where neither the symbolic (t_{Σ}) nor the concrete (t_C) strategy timed out; on the same instance size as Table 1 (for direct comparison); on the largest instance size where at least one strategy did not time out. When two or more lines coincide, they are only listed once.