Model-Based Testing of an Intermediate Verifier Using Executable Operational Semantics*

Lidia Losavio[1**], Marco Paganoni[1**], and Carlo A. Furia[1 [0000−0003−1040−3201]]

Software Institute, USI Università della Svizzera italiana, Lugano, Switzerland
lidia.losavio@usi.ch marco.paganoni@usi.ch bugcounting.net

Abstract. Lightweight validation technique, such as those based on random testing, are sometimes practical alternatives to full formal verification—providing valuable benefits, such as finding bugs, without requiring a disproportionate effort. In fact, such validation techniques can be useful even for fully formally verified tools, by exercising the parts of a complex system that go beyond the reach of formal models.

In this context, this paper introduces BCC: a model-based testing technique for the Boogie intermediate verifier. BCC combines the formalization of a small, deterministic subset of the Boogie language with the generative capabilities of the PLT Redex language engineering framework. Basically, BCC uses PLT Redex to generate random Boogie programs, and to execute them according to a formal operational semantics; then, it runs the same programs through the Boogie verifier. Any inconsistency between the two executions (in PLT Redex and with Boogie) may indicate a potential bug in Boogie's implementation.

To understand whether BCC can be useful in practice, we used it to generate three million Boogie programs. These experiments found 2% of cases indicative of completeness failures (i.e., spurious verification failures) in Boogie's toolchain. These results indicate that lightweight analysis tools, such as those for model-based random testing, are also useful to test and validate formal verification tools such as Boogie.

1 Introduction

Modern verification tools are complex pieces of software; as such, they may contain defects of various kinds that can affect their soundness, precision, performance, or usability. Even if formally verifying a verifier's implementation is the ideal end goal, it usually remains a daunting challenge that requires massive amounts of expert effort [29,10]. This motivates (also) developing *lightweight* techniques [5,31,4,9,46], which do not provide absolute correctness guarantees but can still find errors or provide partial validation with a high degree of automation.

In this vein, this paper describes BCC:[1] a lightweight technique to test the implementation of the Boogie intermediate verifier [3]. BCC relies on a formalization of BPL_0—a small, deterministic subset of the Boogie language—encoded using the PLT Redex

* Work partially supported by SNF grant 200021-207919 (LastMile).

** These authors contributed equally; they are listed in alphabetical order.

[1] BCC stands for "Boogie consistency checker".

framework [11]. In a nutshell, the formal model describes the syntax of BPL$_0$, its typing rules, and its execution semantics by means of symbolic reduction rules. BCC first uses the formal model in PLT Redex to generate random BPL$_0$ programs. Then, it executes the programs according to their operational semantics, and also runs the Boogie verifier on them. The outcome of Boogie's verification run and of the execution in PLT Redex should be *consistent*; BCC reports any inconsistency as a potential *error* of Boogie .[2]

Since it is a best-effort validation technique, BCC cannot provide absolute guarantees of correctness. In particular, Boogie's verification semantics and BCC's operational semantics may be inconsistent in certain cases without implications for correctness: for example, executing a loop in BCC's semantics may time out, whereas Boogie analyzes loops symbolically without checking whether they terminate. Reconciling Boogie's verification semantics and BCC's executable operational semantics is a key challenge addressed in this work—the first, to our knowledge, to apply PLT Redex's semantic engineering techniques to a verification language (as opposed to a conventional programming language).

Our experiments demonstrate that BCC can be practically useful as a bug-finding and validation tool. We generated three million syntactically different random Boogie programs of different sizes and characteristics. BCC found 65 332 cases (2% of all generated programs) of *completeness* failures. Even though BCC only covers a small subset of the whole Boogie language (see Fig. 1), our approach shows that semantics engineering techniques, such as those made available by PLT Redex, can be applied to test formal verification tools, and help improve their reliability.

In summary, the paper makes the following main contributions: *i*) An executable operational semantics of BPL$_0$, a deterministic subset of the Boogie intermediate verification language, built using the PLT Redex framework. *ii*) BCC: a technique to automatically generate Boogie programs with different characteristics, and test their operational semantics against Boogie's . *iii*) A large experimental evaluation of BCC, which found completeness failures in the latest Boogie version. *iv*) For reproducibility, our implementation of PLT Redex and all experimental artifacts are available [30].

2 Related Work

The implementation of a modern formal verification tool is usually a complex piece of software that integrates several independently-developed components. As such, a full end-to-end verification has remained, so far, beyond the capabilities of verification technology. The work that is perhaps closest to the ideal of a fully verified verifier are fully verified *compilers* [27,26,23,43]. Interestingly, even a fully verified compiler like CompCert still benefits from systematic testing [33].

Similarly to a compiler, a formal verification tool usually combines a front-end and a back-end. It is also increasingly common for verifiers to use an intermediate verification language—supported by tools like Boogie [3], Viper [34], and Why3 [14]—which helps bridge the semantic gap between front-end and back-end. There exist a few cases of formally verified back-end provers [39,2,16] and even intermediate verification condition

[2] In principle, an inconsistency may also be due to a bug in the operational semantics rules; in practice, BPL$_0$ and its semantics are so much simpler than Boogie that it's overwhelmingly much more likely that it's Boogie's implementation that at fault.

generators [19]. Several works prove "once and for all" that their front-end *translation* is sound [45,1,40,13], either just on paper or with a mechanized proof. However, none of these works extend the soundness proofs to the actual software *implementations* of the front-end translation.[3] A more viable approach than "once and for all" verification is *run validation*: for any given run of a verifier, extract a checkable *certificate* that the translation is sound. This idea has been applied to validate Boogie's verification condition generation [37], Viper's front-end encoding into Boogie [36], and verifiers based on the K framework [29].

Even in the rare cases where a full formal validation of a verification tool is feasible, *lightweight* tools—usually based on testing—offer a different, appealing trade-off: while they cannot provide absolute guarantees of correctness, they are practically useful to detect bugs, and to test for properties, such as completeness/precision, robustness, or scalability, that are less amenable to formal "all or nothing" verification. In recent years, techniques based on testing have been applied to a variety of formal verification tools, including SMT solvers [31,4,46], intermediate verification languages [9], software model checkers [48,15,44], symbolic execution engines [20], and verifiers for reactive systems [42].

Effectively testing a verification tool requires generating highly structured inputs and complex "behavioral" oracles. To this end, it is common to use grammar-based generation and differential testing—two techniques mutuated from the prolific work on compiler testing [47,24,7], which presents similar challenges. The work presented in this paper also employs these techniques within the PLT Redex framework [11]: a "domain-specific language for semantic models that is embedded in the Racket programming language" [21]. To our knowledge, all applications of PLT Redex to date have been to analyze domain-specific or general-purpose programming languages—often with a functional flavor, such as JavaScript [18] and Lua [41]. One of this paper's contributions is demonstrating how PLT Redex is also applicable to test the imperative features of an intermediate language for verification.

Using a combination of constraint solving and concrete enumeration, previous work targeted executing the Boogie language, with the goal of help debug failed verification attempts [38,28]. Such a goal is largely complementary to the present paper's: whereas those tools explore as efficiently as possible a subset of the (unbounded) execution space of a given (nondeterministic) Boogie program, our goal is checking the consistency of Boogie's verification semantics with a concrete execution semantics on a large number of randomly generated Boogie programs.

3 Modeling a Deterministic Subset of the Boogie Language

BCC deploys a formal model of a deterministic subset of the Boogie intermediate verification language [3], encoded using the PLT Redex framework [11]. The BPL_0 model

[3] In a similar vein, there has been work on formally proving the correctness of verification-condition generation algorithms [35,32,17]—often as part of developing larger verified systems [22]—in a way that a correct-by-construction implementation can be synthesized from the correctness proof. While such an approach can produce trustworthy verification components, it solves a different problem than verifying an existing implementation. For example, a correct-by-construction reimplementation of Boogie would not be a perfect replacement unless it offered the very same performance, features, and capabilities as the actual Boogie tool.

Program P ::=	$(\texttt{main}\ L\ B)$	`procedure main() returns() {`$L\ B$`}`
Locals L ::=	$\emptyset \mid (\texttt{let}\ (v\ \texttt{=}\ \ell\ \texttt{:}\ t)\ L)$	$\epsilon \mid \texttt{var}\ v\texttt{:}\ t := \ell\texttt{;}\ L$
Type t ::=	$\texttt{bool} \mid \texttt{int}$	$\texttt{bool} \mid \texttt{int}$
Body B ::=	$\emptyset \mid (\texttt{do}\ s\ B)$	$\epsilon \mid s$
Statement s ::=	$(\texttt{:=}\ v\ e) \mid (\texttt{if}\ e\ B\ B) \mid (\texttt{while}\ e\ B)$	$v := e \mid \texttt{if}\ e\ B\ \texttt{else}\ B \mid \texttt{while}\ e\ B$
	$\mid (\texttt{assert}\ e)$	$\mid \texttt{assert}\ e$
Expression e ::=	$\ell \mid v \mid (b\ e\ e) \mid (u\ e)$	$\ell \mid v \mid e\ b\ e \mid u\ e$
Literals ℓ ::=	$\mathbb{Z} \mid \texttt{true} \mid \texttt{false}$	
Binary Operator b ::=	$\vee \mid \wedge \mid \Longrightarrow \mid + \mid - \mid / \mid *$	
Unary Operator u ::=	$\neg \mid -$	

Fig. 1: The syntax of BPL$_0$ in PLT Redex (left), and the corresponding Boogie syntax (right).

consists of three components: *i*) a **grammar** defining the **syntax** of *well-formed* BPL$_0$ program terms (Sec. 3.2); *ii*) an **operational semantics** consisting of reduction rules that specify valid executions as transitions between program terms (Sec. 3.3); *iii*) **judgments** consisting of rules for name resolution and type checking (Sec. 3.4).

3.1 Why a *Deterministic* Subset of Boogie?

Before delving into the details of our PLT Redex model of BPL$_0$, let's explain why we focused on this specific restricted subset of the Boogie language. Boogie combines imperative features (e.g., variables) and specification features (e.g., preconditions). Many of Boogie's features introduce nondeterminism, which is widely useful for specification, but is not readily executable. For example, the statement `assume x > 0` is a passive statement that captures all runs where variable `x` is positive. "Executing" the `assume` would mean either enumerating values of `x` up to a finite bound or defining a symbolic semantics. While this different application of PLT Redex could also be interesting, our present goal is to model Boogie's semantics in a way that is straightforward and independent of Boogie's implementation details. This increases the confidence that our execution model is accurate, which bolsters its usefulness as an oracle for differential testing of Boogie. Therefore, BPL$_0$ drops all nondeterministic constructs of Boogie.

3.2 Syntax

Fig. 1 shows the grammar defining the syntax of BPL$_0$. Precisely, Boogie's actual syntax is shown on the right, but in most of the paper we will use PLT Redex's parenthesized prefix notation (since PLT Redex is implemented in Racket) shown on the left.

A program P consists of a single **main** procedure $(\texttt{main}\ L\ B)$ with a local environment L and a body B. A *local environment* is an inductive list of **let**s $(\texttt{let}\ (v\ \texttt{=}\ \ell\ \texttt{:}\ t))$, each binding a variable symbol v to a literal value ℓ of type t (`int` or `bool`). A *body* is an inductive list of statements s: *assignments* $(\texttt{:=}\ v\ e)$, *assertions* $(\texttt{assert}\ e)$, *conditionals* $(\texttt{if}\ e\ B_1\ B_2)$, and *loops* $(\texttt{while}\ e\ B)$. The rest of the grammar defines the usual Boolean and arithmetic *expressions* used in assignments and conditions.

$$
\begin{array}{rll}
E[(b\ \ell_1\ \ell_2)] & \hookrightarrow E[[\![b\ \ell_1\ \ell_2]\!]] & \text{Binary Evaluation} \\
E[(u\ \ell_1)] & \hookrightarrow E[[\![u\ \ell_1]\!]] & \text{Unary Evaluation} \\
(\textbf{main}\ L\ \emptyset) & \hookrightarrow \texttt{success} & \text{Success} \\
E[(\textbf{assert false})] & \hookrightarrow \texttt{failure} & \text{Failure} \\
E[(\textbf{do}\ (\textbf{assert true})\ B)] & \hookrightarrow E[B] & \text{Assert True} \\
(\textbf{main}\ L\ E[v]) & \hookrightarrow (\textbf{main}\ L\ E[L\{v\}]) & \text{Local Substitution} \\
(\textbf{main}\ L\ E[(\textbf{do}\ (:=\ v\ \ell)\ B)]) & \hookrightarrow (\textbf{main}\ L\{v \leftarrow \ell\}\ E[B]) & \text{Local Assignment} \\
E[(\textbf{do}\ (\textbf{if true}\ B_1\ B_2)\ B_3)] & \hookrightarrow E[(\textbf{do}\ B_1 \cdot B_3)] & \text{If-Then} \\
E[(\textbf{do}\ (\textbf{if false}\ B_1\ B_2)\ B_3)] & \hookrightarrow E[(\textbf{do}\ B_2 \cdot B_3)] & \text{If-Else} \\
E[(\textbf{do}\ (\textbf{while}\ e\ B_1)\ B_2)] & \hookrightarrow E[(\textbf{do}\ (\textbf{if}\ e\ B_1 \cdot (\textbf{while}\ e\ B_1)\ \emptyset)\ B_2)] & \text{Loop}
\end{array}
$$

Fig. 2: Operational semantics of BPL$_0$ using evaluation contexts.

$$
\begin{array}{rl}
E ::= & \texttt{hole} \mid (u\ E) \mid (b\ E\ e) \mid (b\ l\ E) \\
\mid & (:=\ v\ E) \mid (\textbf{assert}\ E) \mid (\textbf{if}\ E\ B_1\ B_2) \\
\mid & (\textbf{do}\ E\ B) \mid (\textbf{main}\ L\ E)
\end{array}
$$

Fig. 3: Specification of the possible evaluation contexts for Fig. 2's rules.

Even before formally presenting its semantics, it should be clear that BPL$_0$ is a strictly deterministic language: *i*) each program has a single entry point to the sole **main** procedure; *ii*) before the body executes, the local environment initializes every program variable to a literal value; *iii*) the body's statements execute sequentially; *iv*) all available statements are deterministic.

3.3 Operational Semantics

Fig. 2 shows BPL$_0$'s small-step operational semantics. Each rule defines a different case of the *reduction relation* $A \hookrightarrow B$, which specifies how a program term A rewrites into another one B in a way that captures a single evaluation step.

Most of the reduction rules in Fig. 2 involve an *evaluation context* E. In a nutshell, $E[p]$ means that the program term p can only be reduced (as specified by the rule), when it is in certain parts of the whole program term. Fig. 3 lists all possible evaluation contexts, defined in a way that ensures that the reduction rules can only be applied in a fixed, unambiguous order, which matches the program's sequential order. In particular, the patterns E are defined in such a way that the `hole` can only match in one position.

Fig. 2's rules do not reduce the environment L directly, but use it to keep track of the current values of the program variables in each step of the reduction: $L\{v\}$ denotes a lookup of v's current value in L, whereas $L\{v \leftarrow \ell\}$ denotes an environment where v is bound to value ℓ, and all other variables are as in L.

Most of the meaning of the (other) rules should also be straightforward, as they simply capture: *i*) the evaluation of binary and unary operators, which are simply reduced to the evaluation $[\![\cdot]\!]$ of the corresponding Racket operators and literals; *ii*) success-

ful program termination, when all statements have been "consumed" without errors; *iii*) program termination with a `failure`, when an **`assert`** evaluates to false; *iv*) in contrast, an **`assert`** that passes is equivalent to a skip; *v*) a conditional reduces to its "then" or "else" branch depending on what Boolean the condition evaluates to; *vi*) a loop recursively reduces to evaluating its body until its condition no longer holds.

$$\text{Comp.}\ \frac{L \vdash e_1 : \texttt{int} \quad L \vdash e_2 : \texttt{int}}{L \vdash (\{<, >, =, \leq, \geq\}\ e_1\ e_2) : \texttt{bool}}$$

$$\text{Bin. } \texttt{int}\ \frac{L \vdash e_1 : \texttt{int} \quad L \vdash e_2 : \texttt{int}}{L \vdash (\{+, -, *, /\}\ e_1\ e_2) : \texttt{int}}$$

$$\text{Bin. } \texttt{bool}\ \frac{L \vdash e_1 : \texttt{bool} \quad L \vdash e_2 : \texttt{bool}}{L \vdash (\{\wedge, \vee, \implies, =\}\ e_1\ e_2) : \texttt{bool}}$$

$$\text{Un. } \texttt{int}\ \frac{L \vdash e : \texttt{int}}{L \vdash (-\ e) : \texttt{int}}$$

$$\text{Un. } \texttt{bool}\ \frac{L \vdash e : \texttt{bool}}{L \vdash (\neg\ e) : \texttt{bool}}$$

$$\text{Vars}\ \frac{(v : t) \in L}{L \vdash v : t}$$

$$\text{Assign}\ \frac{L \vdash v : t \quad L \vdash e : t \quad L \vdash B}{L \vdash (\textbf{\texttt{do}}\ (\texttt{:=}\ v\ e)\ B)}$$

$$\text{Assert}\ \frac{L \vdash e : \texttt{bool} \quad L \vdash B}{L \vdash (\textbf{\texttt{do}}\ (\textbf{\texttt{assert}}\ e)\ B)}$$

$$\text{If}\ \frac{L \vdash e : \texttt{bool} \quad L \vdash B_1, B_2, B_3}{L \vdash (\textbf{\texttt{do}}\ (\textbf{\texttt{if}}\ e\ B_1\ B_2)\ B_3)}$$

$$\text{While}\ \frac{L \vdash e : \text{bool} \quad L \vdash B_1, B_2}{L \vdash (\textbf{\texttt{do}}\ (\textbf{\texttt{while}}\ e\ B_1)\ B_2)}$$

$$\text{Empty program}\ \frac{}{L \vdash \emptyset}$$

$$\text{Env.}\ \frac{\emptyset \vdash \ell : t \quad v \notin L \quad \vdash L}{\vdash (\textbf{\texttt{let}}\ [v = \ell : t]\ L)}$$

$$\text{Main}\ \frac{\vdash L \quad L \vdash B}{\vdash (\textbf{\texttt{main}}\ L\ B)}$$

Fig. 4: Inference rules for BPL$_0$'s type system in the PLT Redex model.

3.4 Judgments

BCC can generate *well-formed* BPL$_0$ programs using Fig. 1's grammar. A well-formed program may be invalid because the grammar alone does not ensure that the program is also *well-named* (variables are declared and initialized before their first usage) and *well-typed* (expression values conform to their expected `int` or `bool` type). BCC models BPL$_0$'s name resolution and typing rules using PLT Redex's judgments [12], which can be used by the generation algorithm to automatically produce BPL$_0$ programs that are well-named and/or well-typed.

Fig. 4 presents BPL$_0$'s type system, where $L \vdash e\colon t$ denotes, as customary, that term e has type t when evaluated within local environment L, whereas $L \vdash s$ means that statement s is well-typed. Notice that rules *Vars* and *Env.* also determine whether a program is *well-named* by checking that every used variable was declared and initialized (*Vars*), and that no variable is declared twice (*Env.*). In practice, BCC includes two sets of judgments: the one in Fig. 4 checks well-typedness as well as well-namedness; and another one that *only* checks well-namedness.

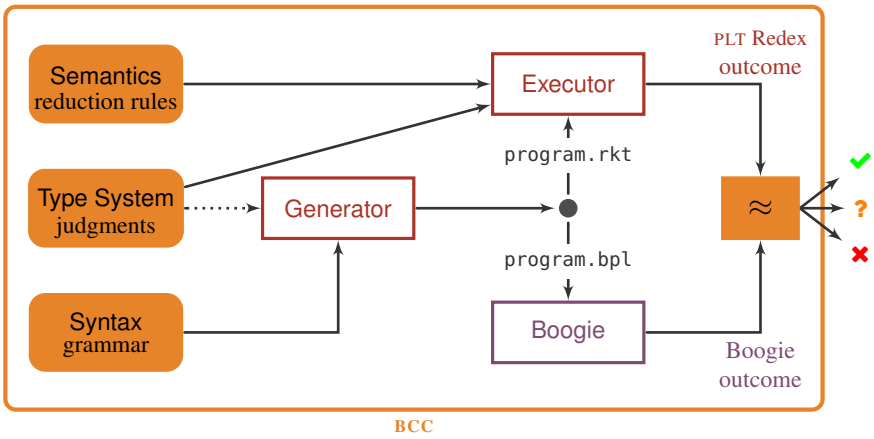

Fig. 5: An overview of how BCC works.

4 How BCC Works

Fig. 5 overviews how BCC uses the PLT Redex model of BPL_0, described in Sec. 3, to: *i*) *Generate* a large number of BPL_0 programs; *ii*) *Execute* them according to their operational semantics; *iii*) *Verify* them using the Boogie verifier; *iv*) *Compare* the outcome of execution and verification, reporting any inconsistency that may be indicative of a failure. The following sections describe each step of BCC's analysis.

4.1 Generator

BCC uses PLT Redex's term generation capabilities to produce a large number of BPL_0 programs. As explained in Sec. 3.4, PLT Redex can produce three kinds of programs: *i*) *well-formed* programs, which conform to BPL_0's grammar; *ii*) *well-named* programs, which are well-formed and also free from any usage of uninitialized variables; *iii*) *well–typed* programs, which are well-named and also typecheck correctly. Each kind of program is suitable to test different aspects of Boogie: its verification process (well-typed), its typechecker (well-named), or its name resolver (well-formed).[4]

BCC's generator actually produces each program in two forms: one uses Racket syntax, and is used by the PLT Redex *Executor*; the other one uses Boogie's concrete syntax, and is fed to *Boogie* itself.

4.2 Executor

Given a BPL_0 program P, BCC first uses the model's *judgments* to check whether P is well-named and well-typed. Obviously, these checks are only necessary if P was not already generated in a well-typed form by construction. In principle, BCC could execute P without performing these checks beforehand, so that any name resolution or type error would result in a runtime failure. However, performing name resolution and type checking *before* executing a program is consistent with Boogie's behavior: it does not

[4] In this work, we did not experiment with generating syntactically incorrect Boogie programs, which would only test Boogie's parser. To this end, there are a number of other techniques, such as fuzzing, that do not require a formal model of Boogie's type system and semantics.

```
procedure success() {
  var x: int := 0;
  assert x = 0;
}
```

```
procedure failure() {
  var x: int := 3;
  assert x < 0;
}
```

```
procedure name_error() {
  var x: int := 0;
  assert y = 0;
}
```

```
procedure type_error() {
  var x: int := 0;
  assert x;
}
```

```
procedure loop() {
  var x: int := 3;
  while (true) {
    x := 0;
  }
  assert false;
}
```

```
procedure timeout() {
  var x: int := 0;
  while (x < 1000000) {
    x := x + 1;
  }
  assert false;
}
```

Table 1: Examples of Boogie programs with various execution and verification outcomes.

run P unless it typechecks. In this way, BCC's execution of P is directly comparable to Boogie's verification run on P.

If P the judgment checks fail, BCC reports one of the following outcomes:

name-error: P is not well-named; for example, program `name_error` in Tab. 1 results in a *name-error* because variable `y` is used but not declared.

type-error: P is not well-typed; for example, program `type_error` in Tab. 1 results in a *type-error* because `int` variable `x` cannot be used as a `bool` assertion.

If P is well-typed, BCC *executes* it by repeatedly applying Fig. 2's reduction rules until one of the following outcomes is reached:

success: P evaluates to an empty body $\emptyset$, which means that the program executed and terminated without any errors; for example, program `success` in Tab. 1 clearly terminates with a *success*.

failure: P evaluates to (`assert false`), which means that the program triggered an assertion failure at some point during its execution; for example, program `failure` in Tab. 1 ends with an assertion *failure* because $\texttt{x} = 3 > 0$.

loop: P evaluates to a term T, which is identical to a term produced during a previous step of the evaluation; this means that the program entered an infinite loop. For example, program `loop` in Tab. 1 results in an infinite loop where the program state does not change. Programs that result in outcome *loop* are still (partially) correct, in that they do not trigger assertion failures: the `assert false` in program `loop` never gets executed because it is effectively unreachable.

timeout: P's evaluation reaches a maximum number of steps (by default, 100 000) and is forcefully terminated; for example, program `timeout` in Tab. 1 exceeds the default maximum number of steps, and hence it would be terminated by BCC. As usual, a *timeout* is inconclusive as to whether P is correct or not: program `timeout` would eventually trigger an assertion failure if its loop were run to completion; but a variant of *timeout* with loop condition `true` would instead never terminate (`int`s are unbounded in Boogie).

4.3 Boogie

BCC also feeds each generated program P to the Boogie verifier, which first checks whether P is well-named and well-typed; if it is, Boogie performs a deductive static verification of P by expressing its correctness conditions as logic formulas, whose validity is checked by the automated theorem prover Z3.

Thus, a Boogie verification run on P reports one of the following outcomes:

name-error: P fails name resolution; for example program `name_error` in Tab. 1.

type-error: P fails type checking; for example program `type_error` in Tab. 1.

success: P verifies correctly, which means that all possible executions of P are free from assertion failures; for example, program `success` in Tab. 1. Boogie does not check termination of loops, but only partial correctness; therefore, program `loop` leads to a *success* in Boogie, because it recognizes that the `assert false` is effectively unreachable.

failure: P fails verification, which means that some assertions in P may not hold; for example, program `failure` in Tab. 1.

timeout: Boogie's verification run on P is forcefully terminated after it does not return within a timeout (by default, one hour). Boogie's timeouts are quite rare on the kinds of programs generated by BCC (which usually verify in a matter of seconds), but sometimes Boogie loses control of the Z3 process it started and does not return within a reasonable amount of time. As usual, we treat a *timeout* as an inconclusive verification result.

Outcome *loop* is inapplicable to Boogie's verification: since it does not analyze termination, it won't report if a loop iterates forever. Finally, a Boogie verification run may also result in a parsing error. Since all the programs generated by BCC are (at least) syntactically correct, this outcome is irrelevant for our work.

PLT Redex outcome p	Boogie outcome b				
	success	*failure*	*timeout*	*name-error*	*type-error*
success	✔	✖	?	✖	✖
failure	✖	✔	?	✖	✖
loop	✔	✖	?	✖	✖
timeout	?	?	?	✖	✖
name-error	✖	✖	✖	✔	✖
type-error	✖	✖	✖	✖	✔

Table 2: Consistency between PLT Redex execution and Boogie verification outcomes.

4.4 Consistency Checking

Finally, BCC compares the outcome of PLT Redex execution according to the operational semantics and Boogie verification for the same program P, and reports any *inconsistency*. This step corresponds to box $\approx$ in Fig. 5.

Tab. 2 shows what BCC's consistency check reports for each combination of PLT Redex outcome p and Boogie outcome b.

✔ denotes that p and b are consistent; hence, we have successfully tested Boogie's correct behavior on a program P.
✖ denotes that p and b are inconsistent; hence, we have exposed a failure in Boogie's behavior, which is inconsistent with the expected semantics of P.
? denotes an inconclusive test: p and b are not directly comparable, and hence Boogie's behavior may or may not be correct.

As shown in Tab. 2, if p and b are literally the same conclusive outcome (*success* or *failure*) then the consistency check obviously results in ✔. However, also the combination $p = \textit{loop}$ and $b = \textit{success}$ is consistent (✔): since *loop* denotes a provably infinite loop, Boogie should also be able to verify that the program in question is (partially) correct, in that any code up to and until the loop is free from errors, whereas the code after the loop is unreachable, and hence irrelevant for correctness.

In contrast, if $p = \textit{failure}$ but $b = \textit{success}$, we would conclude that BCC has found a *soundness* failure ✖, since Boogie erroneously passed as correct an incorrect program (a false negative). Conversely, if $p = \textit{success}$ but $b = \textit{failure}$, we would conclude that BCC has found a *completeness* failure ✖, since Boogie reported a spurious verification failure for a correct program (a false positive); as explained above, the same conclusion ✖ holds if $p = \textit{loop}$ but $b = \textit{failure}$.

Clearly, if $p = \textit{name-error}$—that is, the program is not name-correct—Boogie should report the same outcome, in which case BCC reports consistency ✔; any other Boogie outcome would result in ✖, and reveal that Boogie's name resolution pass has a bug. Similarly, if $p = \textit{type-error}$—that is, the program is not type-correct—Boogie should report the same outcome, unless its type checker has a bug.

Finally, if $p = \textit{timeout}$, the consistency check is inconclusive (?) regardless of whether $b = \textit{success}$, $b = \textit{failure}$, or $b = \textit{timeout}$; similarly, if $b = \textit{timeout}$, the consistency check is inconclusive (?) regardless of whether $p = \textit{success}$, $p = \textit{failure}$, or $p = \textit{loop}$. For example, as discussed previously, program `timeout` in Tab. 1 results in $p = \textit{timeout}$ and $b = \textit{failure}$, which is consistent because the program is actually incorrect; however, if we removed the `assert false`, PLT Redex's outcome p would still be *timeout* but Boogie would correctly report $b = \textit{success}$ because the program would now be correct.

```
while (C)                    while (C)
  invariant J {                invariant J {
  // ...                         assert A;
}                                // ...
assert A;                    }
```

Fig. 6: Generic Boogie loops annotated with loop invariants.

Loop invariants. To properly gauge the inconsistencies in our experiments that point to *completeness* failures of Boogie, we need to understand the role played by loop *invariants*. Consider Fig. 6's generic loops (annotated with an invariant J), and suppose that Boogie's outcome b is a *failure* of `assert` A, whereas PLT Redex reports $p =$ *loop* (program on the left) or $p =$ *success* (program on the right) on the same program—hence, the program is (partially) correct. For the program on the left, if $\neg C \Rightarrow A$ holds we say that it is a case of *reasoning* incompleteness of Boogie; otherwise, we call it a case of *annotation* incompleteness. Similarly, for the program on the right, if $\neg C$ holds at the loop's entrance we say that it is a case of *reasoning* incompleteness; otherwise, we call it a case of *annotation* incompleteness. In principle, annotation incompleteness merely indicates that the user should have annotated the loop with an invariant strong enough to characterize the loop's behavior: if J is indeed a correct invariant of the loop, then Boogie could use it to prove that $J \wedge \neg C \Rightarrow A$ (program on the left) or $J \Rightarrow \neg C$ (program on the right), hence establishing that the program is correct after all. Conversely, reasoning incompleteness means that A holds independent of any loop invariants, because it is a direct consequence of the loop's condition C; hence, Boogie should be able to successfully verify the program without additional annotations. These definitions generalize to cases where the assertion that spuriously fails in Boogie is not immediately after the loop (resp. at the beginning of the loop body) but follows a sequence S of statements.

Distinguishing between the two kinds of incompleteness is undecidable in general, because it requires a sound and complete loop invariant inference. This also explains why BPL$_0$ does not include loop invariant annotations: synthesizing correct (and useful) loop invariants for randomly generated loops is nontrivial, and would require to introduce different techniques. Sec. 5.3 describes how we tried to estimate annotation vs. reasoning incompleteness in our experiments.

5 Experiments

To evaluate BCC's capabilities, we used it to generate a large number of BPL$_0$ programs with different characteristics, and to compare the outcome of PLT Redex execution and Boogie verification on those programs.

5.1 Setup

Batches. Each *batch* of generated programs is defined by three parameters: *i*) the *number* of BPL$_0$ programs in the batch; *ii*) whether they should be well-*typed*, well-*named*, or simply well-*formed*; *iii*) the maximum *size* of each generated program.[5] In our experiments, we generated a total of 3 million syntactically different programs in 12 batches: for each kind of well-typed, well-named, and well-formed programs, *i*) 100 000 programs with maximum size 3, *ii*) 200 000 programs with maximum size 5, *iii*) 200 000 programs with maximum size 7, *iv*) 500 000 programs with maximum size 10. We chose these numbers after some informal experiments with BCC, which suggested two guidelines: first, even though a larger maximum program size should permit also small programs, using batches with different maximum sizes achieves a better program size

[5] PLT Redex's documentation is somewhat vague about how this `size-expr` parameter is used, but it's probably an upper bound on the maximum depth of the generated term tree.

Batch	Size	Locals L			Statement s			Expr e (arith)			Expr e (bool)			Expr e (comp)			Literals l		
		min	med	max	min	med	max	min	med	max	min	med	max	min	med	max	min	med	max
formed	3	1	1	3	1	9	18	0	1	5	0	4	14	0	0	5	0	7	16
	5	1	1	5	1	20	65	0	2	14	0	10	37	0	1	11	0	15	51
	7	1	1	7	1	40	176	0	5	39	0	20	110	0	2	22	0	31	132
	10	1	2	10	1	100	637	0	16	126	0	54	379	0	8	78	0	79	517
named	3	1	1	3	1	10	18	0	0	5	0	5	17	0	0	5	0	10	20
	5	1	1	5	1	21	61	0	2	15	0	12	41	0	1	11	0	20	57
	7	1	1	7	1	41	130	0	4	31	0	23	106	0	2	19	0	38	125
	10	1	1	10	1	102	491	0	14	108	0	60	326	0	7	66	0	93	471
typed	3	1	1	3	1	10	18	0	0	4	0	7	27	0	1	6	0	8	20
	5	1	1	5	1	21	54	0	3	17	0	17	65	0	3	16	0	18	58
	7	1	1	7	1	41	139	0	9	48	0	35	187	0	7	49	0	38	170
	10	1	2	10	1	103	481	0	37	265	0	94	553	0	24	157	0	106	584

Table 3: Statistics about the composition of the BPL$_0$ programs generated in the experiments. For each batch of *well-formed*, *well-named*, or *well-typed* programs of a given maximum *size*, the table reports *min*imum, *med*ian, and *max*imum number of terms of each kind in any program in the batch.

diversity; second, batches with a smaller maximum program size should be smaller, otherwise there is a risk of inefficiently generating a lot of nearly identical programs.

System. The experiments ran on a virtual machine with 64 cores of an AMD Epyc 7713 3.7 Ghz processor with 128 GB of RAM, running Ubuntu 22.04, Racket/PLT Redex v. 8.2, Boogie v. 3.4.3 with Z3 v. 4.8.8—the latest versions of the software at the time of these experiments. To speed up the experiments, several instances of BCC ran in parallel processes, each on different BPL$_0$ programs and coordinated by a work-stealing algorithm; precisely, the generation phase used 64 parallel processes, the execution phase 128, and the verification phase 512.

5.2 Results: Generation, Execution, and Verification

Generation In our experiments, BCC took about 30 minutes to generate the 3 million programs in 12 batches. Tab. 3 shows basic statistics about the actual size of the BPL$_0$ programs generated in each batch, measured in terms of the number of local variables, statements, arithmetic, Boolean and comparison operators, and literals that appear in a program. Most programs have a non-trivial size in terms of statements and expressions, whereas they tend to include few local variables (median: 1–2, maximum: 10). Well-typed programs are about as large, on average, as well-named and well-formed programs, but they tend to include larger expressions. This is an indirect effect of using typing judgments as a constraint to generate *well-typed* programs: this additional condition nudges the generation to produce more "interesting", deeply nested expressions; in contrast, the generation of well-formed programs has fewer constraints, and hence results in a more shallow enumeration of all possible literal/operator combination. Overall, BCC managed to generate a broad range of programs with different sizes and characteristics.

Execution In our experiments, BCC took around 110 minutes to execute the 3 million programs according to BPL$_0$'s operational semantics. Tab. 4's left half shows the number of programs in each batch whose PLT Redex execution resulted in one of five possible

Batch	Size	PLT Redex Execution Outcomes						Boogie Verification Outcomes				
		success	*failure*	*loop*	*timeout*	*name-error*	*type-error*	*success*	*failure*	*timeout*	*name-error*	*type-error*
formed	3	2	0	0	0	99 617	381	2	0	0	99 617	381
	5	2	2	0	0	199 917	79	2	2	0	199 917	79
	7	6	3	1	0	199 929	61	7	3	0	199 929	61
	10	14	5	1	0	499 838	142	15	5	0	499 838	142
all	#	24	10	2	0	999 301	663	26	10	0	999 301	663
	%	0	0	0	0	100	0	0	0	0	100	0
named	3	23	25	14	0	0	99 938	37	25	0	0	99 938
	5	21	9	1	0	0	199 969	22	9	0	0	199 969
	7	27	6	2	0	0	199 965	29	6	0	0	199 965
	10	73	18	7	0	0	499 902	80	18	0	0	499 902
all	#	144	58	24	0	0	999 774	168	58	0	0	999 774
	%	0	0	0	0	0	100	0	0	0	0	100
typed	3	3 664	64 888	31 446	2	0	0	33 960	66 040	0	0	0
	5	1 513	130 578	67 661	248	0	0	62 126	137 874	0	0	0
	7	675	130 200	68 746	379	0	0	57 021	142 978	1	0	0
	10	1 310	341 674	155 942	1 074	0	0	113 711	386 264	25	0	0
all	#	7 162	667 340	323 795	1 703	0	0	266 818	733 156	26	0	0
	%	1	67	32	0	0	0	27	73	0	0	0
all	#	7 330	667 408	323 821	1 703	999 301	1 000 437	267 012	733 224	26	999 301	1 000 437
	%	0	22	11	0	33	33	9	24	0	33	33

Table 4: Statistics about the outcome of PLT *Redex execution* (left) and *Boogie verification* (right) of the BPL_0 programs generated in the experiments. For each batch of *well-formed*, *well-named*, or *well-typed* programs of a given maximum *size*, the number of programs in the batch whose PLT Redex execution resulted in a certain outcome. Rows *all* aggregate the counts # and percentage % within each batch, and in all dataset.

outcomes: *success*, *failure*, *loop*, *timeout*, *name-error*, and *type-error*. As expected, most well-formed programs resulted in a name resolution error; and most well-named programs in a type checking error (even though a few managed to pass typechecking, and even to pass execution).

As for well-typed programs, less than 1% of them timed out; thus, most executions ran to completion. Similarly, only about 1% of the programs terminated with a *success*; this indicates that it is more likely that a randomly generated assertion fails than it holds. In fact, *failure* was by far the most common outcome of executing well-typed programs: over $2/3$ of them terminated with an assertion failure. Within each batch of a certain maximum size, the percentage of programs that fail is similar (around 65%) but becomes a bit higher for the batch with size 10 (around 68%), arguably because larger random expression are more likely to introduce a contradiction. Nevertheless, PLT Redex detected an infinite *loop* in nearly $1/3$ of the randomly generated programs. In all, despite the simplicity of the Boogie BPL_0 subset it targets, the programs generated by BCC have a good variety of possible outcomes, and thus have the potential of exercising the verifier in different conditions.

Verification In our experiments, BCC took around 12 hours to verify the 3 million programs with Boogie; unsurprisingly, running Boogie is an order of magnitude more time consuming than running PLT Redex on relatively small programs (or generating them),

since each Boogie run usually involves calls to an SMT solver. As during PLT Redex execution, Boogie reports a *name-error* (resp. *type-error*), on most well-formed (resp. well-named) programs.

The distribution of verification outcomes for well-typed programs may seem quite different from that of execution outcomes, but it is actually largely consistent. In fact, remember that both execution outcomes $p = \textit{success}, \textit{loop}$ correspond to the same verification outcome $b = \textit{success}$—since a program with an infinite loop should be classified as correct by Boogie, which only checks partial correctness. Boogie timeouts are exceedingly rare, at least with the kinds of programs that BCC generates, which do not include complex quantified formulas that may trip up the quantifier instantiation algorithms [25,8]. However, Boogie timeouts do occasionally, and unpredictably, happen on programs that, upon inspection, do not seem to have any complex or unusual feature that stands out.

PLT Redex outcome p	Boogie outcome b: *success*	*failure*	*timeout*	*name-error*	*type-error*
success	6 981 0.23%	349 0.01%			
failure		667 397 22.25%	11 0.00%		
loop	258 808 8.63%	64 998 2.17%	15 0.00%		
timeout	1 223 0.04%	480 0.02%			
name-error				999 301 33.31%	
type-error					1 000 437 33.35%

Table 5: Number and percentage of all BPL_0 programs generated in the experiments, for each combination of PLT Redex execution outcome p and Boogie verification outcome b as in Tab. 2. A red background highlights the combinations that correspond to *inconsistent* results that have been observed in the experiments.

5.3 Results: Consistency Checks

Tab. 5 reports the number and percentage of all 3 million programs generated in the experiments that resulted in each of Tab. 2's possible combinations of PLT Redex execution outcome p and Boogie verification outcome b.

Correctness In the vast majority of cases, Boogie's outcome was consistent with the result of executing according to BPL_0's operational semantics. In particular, all programs with name errors or type errors were caught by Boogie's name resolution or type checking modules.

Boogie successfully verified 80% (i.e., $(6\,981 + 258\,808)/(7\,330 + 323\,821)$) of all (partially) correct programs according to PLT Redex execution (that is, p outcomes *success* and *loop*); it also confirmed as incorrect nearly 100% (i.e., $667\,397/667\,408$) of all incorrect programs according to PLT Redex execution (p outcome *failure*). The cases of programs involving timeouts are inconclusive, but they are also only a tiny fraction of the total. Overall, BCC was useful to thoroughly validate Boogie, which behaved correctly in the vast majority of cases.

PLT Redex outcome p	Boogie outcome b *invariant inference* *success*	*failure*	*success*	*failure*
success	7 259	71	6 981	349
loop	309 020	14 785	258 808	64 998

Table 6: How Boogie's inconsistency failures change if it uses loop invariant inference. The two rightmost columns are the same as in Tab. 5. A red background highlights the combinations that correspond to *incompleteness* failures.

Soundness BCC did not expose any soundness failure of Boogie, that is cases of incorrect programs (p = *failure*) that Boogie passes as correct (b = *success*). While it is known that SMT solvers—including Z3—do sometimes suffer from soundness bugs, exposing them requires bespoke constraints [46], which are unlikely to be generated by BCC's grammar during purely random enumeration. Furthermore, Boogie's verification condition generation algorithm adds a layer of indirection between the Boogie program and the constraints sent to the SMT solver, which may complicate triggering soundness bugs through Boogie without involving features, such as triggers, that directly interact with the underlying solver [6]. Despite these limitations, BCC's testing remains useful to increase our confidence that Boogie is generally sound.

Completeness Our experiments found $65\,347$ $(349 + 64\,998)$ cases of *completeness* failure; that is, Boogie rejected as (possibly) incorrect (b = *failure*) about 20% (i.e., $65\,347/(7\,330 + 323\,821)$) of all correct programs (PLT Redex p outcomes *success* or *loop*). These results are largely consistent with how Boogie is designed: since it aims at being a sound implementation of deductive program verification (which is undecidable in general), it will necessarily incur cases of incompleteness. Still, they also further demonstrate BCC's practical usefulness in stress-testing a verifier to better reveal its capabilities and limitations.

Sec. 4.4 introduced the distinction between *annotation* and *reasoning* incompleteness. How many of our experiments triggered each kind of incompleteness? To answer this question, we tried to use Boogie's support for loop invariant inference as a proxy for distinguishing between reasoning and annotation incompleteness. For each program P where b = *failure* but p = *success* or *loop*, we ran Boogie on P again with option `/infer:j`—which enables loop invariant inference. If P verifies successfully in this new run, it suggests that it is a case of annotation incompleteness; otherwise, we classify it as reasoning incompleteness. We stress that this is an imperfect proxy, which is, in general, neither sound nor precise: *i*) obviously, if P still fails verification, it may just mean that Boogie's loop invariant inference could not find the "right" loop invariant; *ii*) conversely, if P passes verification, it may still be a case of reasoning incompleteness, where adding a loop invariant J unexpectedly helps establish that A holds, even if J is subsumed by $\neg C$ or C.

With these caveats in mind, let's have a look at Tab. 6, which shows how the number of incompleteness inconsistencies change if Boogie uses loop invariant inference. The number of inconsistencies shrinks significantly: only 23% (i.e., $(71 + 14\,785)/(349 +$

64 998)) of spurious failures encountered in our experiments remain even if loop invariant inference is enabled. This would seem to suggest that completeness failures often denote annotation incompleteness rather than reasoning incompleteness. On the other hand, a nontrivial number of cases of actual annotation incompleteness seem to remain (those resistant to invariant inference).

In order to better understand the effect of loop invariant inference, we selected a small random sample consisting of 40 Boogie programs among those that PLT Redex execution confirmed as correct (p = *success* or *loop*): 20 among those that Boogie without loop invariant inference flagged as incorrect (b = *failure*), and Boogie with loop invariant inference verified successfully (b = *success*); and 20 among those that Boogie flagged as incorrect (b = *failure*) with or without loop invariant inference. We manually inspected these 40 Boogie programs, trying to determine whether they were cases of annotation or reasoning incompleteness. We found that 23 cases[6] out of 40 were actually instances of *reasoning* incompleteness, since they all involved unreachable code guarded by a condition that was identically false (independent of any additional annotation). Therefore, loop invariant inference should not have any effect on whether Boogie can verify these examples successfully. In practice, option `/infer:j` introduces several nontrivial changes in how Boogie generates the verification conditions of a program; therefore, it is to be expected that it can have an indirect, unpredictable effect on the kinds of programs that can be automatically verified—it is a form of brittleness, also observed in related work [9]. In all, we have reasons to believe that a larger fraction of the completeness failures encountered in BCC's experiments are actually indicative of *reasoning* incompleteness—even though it is hard to get a precise estimate without a very prohibitively time-consuming extensive manual analysis. Regardless, our experiments successfully demonstrated BCC's practical applicability to run large-scale thorough testing of the the Boogie intermediate verifier.

Examples of Reasoning Incompleteness Fig. 7 shows two programs generated by BCC in our experiments[7] that showcase *completeness* failures. For space reasons we only shows these two examples, which are, however, quite representative of numerous other similar examples that exhibit the same behavior.

The outermost loop's body in Fig. 7a's program `alwaysLoops` (which has the same general structure as Fig. 6's left program) clearly never terminates, since variable `G` is initialized to `true` and never changes value. While PLT Redex execution results in p = *loop*, Boogie returns with b = *failure*, flagging a violation of the `assert false` just after the outer loop. According to our classification, this is a case of annotation incompleteness, since Boogie correctly verifies the program if it is given the additional information that `G` is always true (which can be retrieved by loop invariant inference). Note that, in this case, the invariance of `G` is path and flow insensitive: in the loop, there is a single assignment to `G` of the constant `true`.

Conversely, the outermost loop's body in Fig. 7b's program `neverLoops` clearly never executes, since variable `s` is initialized to `false` and immediately checked as the

[6] Precisely, 14 cases that Boogie verifies only with `/infer:j`, and 9 cases that fail verification even with `/infer:j`.

[7] For readability, we simplified the full programs by removing parts that do not affect their behavior.

```
procedure alwaysLoops() returns () {
  var G : bool;
  G := true;
  G := G;
  assert (0 < 1) ∧ (1 = 1);
  while (G) {
    while (G) {
      assert ¬(true ⟹ false);
   }
   G := true;
  }
  //...
  assert false;
}
```

(a) The outer loop in this program never terminates.

```
procedure neverLoops() returns () {
  var s: bool; var AE: bool;
  s := false; AE := false;
  while (s) {
    if ((−0 * 0 + −0) > 1) {
    } else {
      while (s) {
        AE := true;
        s := ¬(s ∧ ¬false);
      }
    }
    s := true;
    if (0 ≥ −3) {
      if (false) {
      } else {
        assert (¬(true ∧ s) = AE);
      }
    }
    /* ... */ } }
```

(b) The outer loop in this program never executes.

Fig. 7: Two programs generated by BCC that expose incompleteness in Boogie.

loop condition. While PLT Redex execution results in $p = \textit{success}$, Boogie returns with $b = \textit{failure}$, flagging a violation of the assertion at the end of the loop body. Since the outer loop condition is obviously identically `false` when it is first evaluated, this is a case of reasoning incompleteness. However, Boogie returns with $b = \textit{success}$ if we enable the `/infer:j` option, even though a loop invariant is not needed to determine that the loop never executes. Even more unpredictably, if we add more statements after the `assert` within the loop body, then Boogie reports a (spurious) verification failure regardless of whether loop invariant inference is or isn't enabled.

6 Conclusions

This paper presented BCC: a lightweight validation technique to test deductive verifiers such as the widely used Boogie. BCC is built atop a formal model of a small, deterministic subset of the Boogie language, consisting of a grammar, typing rules, and an executable operational semantics—encoded using the PLT Redex semantic engineering framework. In our experiments, we used BCC to thoroughly test Boogie for consistency; while we found that Boogie's output is correct and reliable in the vast majority of cases, we did find a few interesting examples that expose completeness failures.

References

1. Backes, M., Hritcu, C., Tarrach, T.: Automatically verifying typing constraints for a data processing language. In: Jouannaud, J., Shao, Z. (eds.) Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings. Lecture Notes in Computer Science, vol. 7086, pp. 296–313. Springer (2011). https://doi.org/10.1007/978-3-642-25379-9_22, https://doi.org/10.1007/978-3-642-25379-9_22
2. Baek, S.: A formally verified checker for first-order proofs. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference). LIPIcs, vol. 193, pp. 6:1–6:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPICS.ITP.2021.6, https://doi.org/10.4230/LIPIcs.ITP.2021.6
3. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17, https://doi.org/10.1007/11804192_17
4. Bringolf, M., Winterer, D., Su, Z.: Finding and understanding incompleteness bugs in SMT solvers. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022. pp. 43:1–43:10. ACM (2022). https://doi.org/10.1145/3551349.3560435, https://doi.org/10.1145/3551349.3560435
5. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6175, pp. 44–57. Springer (2010). https://doi.org/10.1007/978-3-642-14186-7_6, https://doi.org/10.1007/978-3-642-14186-7_6
6. Bugariu, A., Ter-Gabrielyan, A., Müller, P.: Identifying overly restrictive matching patterns in SMT-based program verifiers. In: Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13047, pp. 273–291. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_15, https://doi.org/10.1007/978-3-030-90870-6_15
7. Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L.: A survey of compiler testing. ACM Comput. Surv. **53**(1), 4:1–4:36 (2021). https://doi.org/10.1145/3363562, https://doi.org/10.1145/3363562
8. Chen, Y., Furia, C.A.: Triggerless happy – intermediate verification with a first-order prover. In: Proceedings of the 13th International Conference on integrated Formal Methods (iFM). Lecture Notes in Computer Science, vol. 10510, pp. 295–311. Springer (2017)
9. Chen, Y., Furia, C.A.: Robustness testing of intermediate verifiers. In: Lahiri, S., Wang, C. (eds.) Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA). Lecture Notes in Computer Science, vol. 11138, pp. 91–108. Springer (September 2018)
10. Dardinier, T., Sammler, M., Parthasarathy, G., Summers, A.J., Müller, P.: Formal foundations for translational separation logic verifiers. Proc. ACM Program. Lang. **9**(POPL), 569–599 (2025). https://doi.org/10.1145/3704856, https://doi.org/10.1145/3704856
11. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press (2009), http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885
12. Fetscher, B., Claessen, K., Palka, M.H., Hughes, J., Findler, R.B.: Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In:

Vitek, J. (ed.) Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9032, pp. 383–405. Springer (2015). https://doi.org/10.1007/978-3-662-46669-8_16, https://doi.org/10.1007/978-3-662-46669-8_16
13. Fiala, J., Itzhaky, S., Müller, P., Polikarpova, N., Sergey, I.: Leveraging rust types for program synthesis. Proc. ACM Program. Lang. **7**(PLDI), 1414–1437 (2023). https://doi.org/10.1145/3591278, https://doi.org/10.1145/3591278
14. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8, https://doi.org/10.1007/978-3-642-37036-6_8
15. Fink, X., Berger, P., Katoen, J.: Configurable benchmarks for C model checkers. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13260, pp. 338–354. Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_18, https://doi.org/10.1007/978-3-031-06773-0_18
16. From, A.H., Jacobsen, F.K.: Verifying a sequent calculus prover for first-order logic with functions in Isabelle/HOL. J. Autom. Reason. **68**(3), 15 (2024). https://doi.org/10.1007/S10817-024-09697-3, https://doi.org/10.1007/s10817-024-09697-3
17. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: formal verification of C code without the pain. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 429–439. ACM (2014). https://doi.org/10.1145/2594291.2594296, https://doi.org/10.1145/2594291.2594296
18. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of javascript. In: D'Hondt, T. (ed.) ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6183, pp. 126–150. Springer (2010). https://doi.org/10.1007/978-3-642-14107-2_7, https://doi.org/10.1007/978-3-642-14107-2_7
19. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. In: Joshi, R., Müller, P., Podelski, A. (eds.) Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7152, pp. 2–17. Springer (2012). https://doi.org/10.1007/978-3-642-27705-4_2, https://doi.org/10.1007/978-3-642-27705-4_2
20. Kapus, T., Cadar, C.: Automatic testing of symbolic execution engines via program generation and differential testing. In: Rosu, G., Penta, M.D., Nguyen, T.N. (eds.) Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017. pp. 590–600. IEEE Computer Society (2017). https://doi.org/10.1109/ASE.2017.8115669, https://doi.org/10.1109/ASE.2017.8115669
21. Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J.A., Rafkind, J., Tobin-Hochstadt, S., Findler, R.B.: Run your research: on the effectiveness of lightweight mechanization. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. pp. 285–296. ACM (2012). https://doi.org/10.1145/2103656.2103691, https://doi.org/10.1145/2103656.2103691

22. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D.A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010). https://doi.org/10.1145/1743546.1743574, https://doi.org/10.1145/1743546.1743574
23. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 179–192. ACM (2014). https://doi.org/10.1145/2535838.2535841, https://doi.org/10.1145/2535838.2535841
24. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 216–226. ACM (2014). https://doi.org/10.1145/2594291.2594334, https://doi.org/10.1145/2594291.2594334
25. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: CAV. pp. 361–381. LNCS, Springer (2016)
26. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009), http://xavierleroy.org/publi/compcert-backend.pdf
27. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning **41**(1), 1–31 (2008), http://xavierleroy.org/publi/memory-model-journal.pdf
28. Liew, D., Cadar, C., Donaldson, A.: Symbooglix: A symbolic execution engine for Boogie programs. In: IEEE International Conference on Software Testing, Verification, and Validation (ICST 2016). pp. 45–56 (4 2016)
29. Lin, Z., Chen, X., Trinh, M., Wang, J., Rosu, G.: Generating proof certificates for a language-agnostic deductive program verifier. Proc. ACM Program. Lang. **7**(OOPSLA1), 56–84 (2023). https://doi.org/10.1145/3586029, https://doi.org/10.1145/3586029
30. Losavio, L., Paganoni, M., Furia, C.A.: Boogie Consistency Checker: Replication package for iFM2025 (June 2025). https://doi.org/10.6084/m9.figshare.29338589, https://doi.org/10.6084/m9.figshare.29338589.v2
31. Mansur, M.N., Christakis, M., Wüstholz, V., Zhang, F.: Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. pp. 701–712. ACM (2020). https://doi.org/10.1145/3368089.3409763, https://doi.org/10.1145/3368089.3409763
32. Marti, N., Affeldt, R.: A certified verifier for a fragment of separation logic. Inf. Media Technol. **4**(2), 304–316 (2009). https://doi.org/10.11185/IMT.4.304, https://doi.org/10.11185/imt.4.304
33. Monniaux, D., Gourdin, L., Boulmé, S., Lebeltel, O.: Testing a formally verified compiler. In: Prevosto, V., Seceleanu, C. (eds.) Tests and Proofs – 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14066, pp. 40–48. Springer (2023). https://doi.org/10.1007/978-3-031-38828-6_3, https://doi.org/10.1007/978-3-031-38828-6_3
34. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 9583, pp. 41–62. Springer-Verlag (2016), https://doi.org/10.1007/978-3-662-49122-5_2
35. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics. Formal Aspects Comput. **10**(2), 171–186 (1998). https://doi.org/10.1007/S001650050009, https://doi.org/10.1007/s001650050009

36. Parthasarathy, G., Dardinier, T., Bonneau, B., Müller, P., Summers, A.J.: Towards trustworthy automated program verifiers: Formally validating translations into an intermediate verification language. Proc. ACM Program. Lang. **8**(PLDI), 1510–1534 (2024). https://doi.org/10.1145/3656438, https://doi.org/10.1145/3656438
37. Parthasarathy, G., Müller, P., Summers, A.J.: Formally validating a practical verification condition generator. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 704–727. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_33, https://doi.org/10.1007/978-3-030-81688-9_33
38. Polikarpova, N., Furia, C.A., West, S.: To run what no one has run before: Executing an intermediate verification language. In: Proceedings of the 4th International Conference on Runtime Verification (RV). Lecture Notes in Computer Science, vol. 8174, pp. 251–268. Springer (2013)
39. Skotåm, S.H.: CreuSAT, Using Rust and Creusot to create the world's fastest deductively verified SAT solver. Master's thesis, University of Oslo (2022), https://www.duo.uio.no/handle/10852/96757
40. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012). https://doi.org/10.1145/2160910.2160911, https://doi.org/10.1145/2160910.2160911
41. Soldevila, M., Ziliani, B., Silvestre, B., Fridlender, D., Mascarenhas, F.: Decoding Lua: formal semantics for the developer and the semanticist. In: Ancona, D. (ed.) Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017. pp. 75–86. ACM (2017). https://doi.org/10.1145/3133841.3133848, https://doi.org/10.1145/3133841.3133848
42. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation: synthesizing programs of realistic structure. Int. J. Softw. Tools Technol. Transf. **16**(5), 465–479 (2014). https://doi.org/10.1007/S10009-014-0336-Z, https://doi.org/10.1007/s10009-014-0336-z
43. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A.C.J., Owens, S., Norrish, M.: The verified CakeML compiler backend. J. Funct. Program. **29**, e2 (2019). https://doi.org/10.1017/S0956796818000229, https://doi.org/10.1017/S0956796818000229
44. Thoben, N., Haltermann, J., Wehrheim, H.: Timeout prediction for software analyses. In: Ferreira, C., Willemse, T.A.C. (eds.) Software Engineering and Formal Methods - 21st International Conference, SEFM 2023, Eindhoven, The Netherlands, November 6-10, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14323, pp. 340–358. Springer (2023). https://doi.org/10.1007/978-3-031-47115-5_19, https://doi.org/10.1007/978-3-031-47115-5_19
45. Vogels, F., Jacobs, B., Piessens, F.: A machine checked soundness proof for an intermediate verification language. In: Nielsen, M., Kucera, A., Miltersen, P.B., Palamidessi, C., Tuma, P., Valencia, F.D. (eds.) SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 24-30, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5404, pp. 570–581. Springer (2009). https://doi.org/10.1007/978-3-540-95891-8_51, https://doi.org/10.1007/978-3-540-95891-8_51
46. Winterer, D., Su, Z.: Validating SMT solvers for correctness and performance via grammar-based enumeration. Proc. ACM Program. Lang. **8**(OOPSLA2), 2378–2401 (2024). https://doi.org/10.1145/3689795, https://doi.org/10.1145/3689795
47. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference

on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 283–294. ACM (2011). https://doi.org/10.1145/1993498.1993532, https://doi.org/10.1145/1993498.1993532

48. Zhang, C., Su, T., Yan, Y., Zhang, F., Pu, G., Su, Z.: Finding and understanding bugs in software model checkers. In: Dumas, M., Pfahl, D., Apel, S., Russo, A. (eds.) Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019. pp. 763–773. ACM (2019). https://doi.org/10.1145/3338906.3338932, https://doi.org/10.1145/3338906.3338932