THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Towards Robust and Flexible Intermediate Verification

Yu-Ting Chen

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG

Gothenburg, Sweden
2018
Towards Robust and Flexible Intermediate Verification

© 2018 Yu-Ting Chen

Technical Report 176L
ISSN 1652-876X
Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY AND
UNIVERSITY OF GOTHENBURG
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Printed at Reproservice, Chalmers University of Technology
Gothenburg, Sweden, 2018
Abstract

Automated static program verifiers are more accessible than ever before thanks to verification infrastructures like intermediate verification languages (IVLs). The introduction of IVLs such as Boogie and Why3 accelerates the development of program verifiers interfacing different front-end languages. Coupled with recent advents in satisfiability modulo theories (SMT) solvers, fully automated program verification is getting more attention. However, to address these inherently incomplete problems, certain heuristics must be used for provers to discharge proof obligations. Occasionally, elaborated heuristics meant for efficient proof search of one problem genre can lead to obscure proof failures of another simple correctness proof only resolvable by solver-specific experts.

This thesis explores intermediate verification and proposes two contributions to improve practicality: flexible and robust intermediate verification.
# Contents

1 Introduction 3
   1.1 Grand Challenge: Practical Program Verification  . . . . . . . . 4
   1.2 State of the Art  . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   1.3 Contributions  . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   1.4 Conclusion and Future Work  . . . . . . . . . . . . . . . . . . . 9

2 Triggerless Happy: Intermediate Verification with a First-Order Prover 13
   2.1 The Trouble with Triggers  . . . . . . . . . . . . . . . . . . . . . 15
   2.2 Motivating Examples  . . . . . . . . . . . . . . . . . . . . . . . 16
   2.3 Encoding Boogie in TPTP  . . . . . . . . . . . . . . . . . . . . . 19
      2.3.1 Declarative constructs  . . . . . . . . . . . . . . . . . . . . 19
      2.3.2 Imperative constructs  . . . . . . . . . . . . . . . . . . . . 21
   2.4 Implementation and Experiments  . . . . . . . . . . . . . . . . . 25
      2.4.1 Experimental subjects  . . . . . . . . . . . . . . . . . . . . 25
      2.4.2 Experimental setup  . . . . . . . . . . . . . . . . . . . . . 26
      2.4.3 Experimental results  . . . . . . . . . . . . . . . . . . . . 27
   2.5 Related Work  . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
   2.6 Discussion and Future Work  . . . . . . . . . . . . . . . . . . . 30

3 Robustness Testing of Intermediate Verifiers 33
   3.1 Introduction  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
   3.2 Motivating Example  . . . . . . . . . . . . . . . . . . . . . . . . 37
   3.3 How Robustness Testing Works  . . . . . . . . . . . . . . . . . . 38
      3.3.1 Mutation Operators  . . . . . . . . . . . . . . . . . . . . . 39
      3.3.2 Mutation Generation  . . . . . . . . . . . . . . . . . . . . . 41
      3.3.3 Implementation  . . . . . . . . . . . . . . . . . . . . . . . 43
   3.4 Experimental Evaluation  . . . . . . . . . . . . . . . . . . . . . 43
      3.4.1 Experimental Design  . . . . . . . . . . . . . . . . . . . . . 43
      3.4.2 Experimental Results  . . . . . . . . . . . . . . . . . . . . 47
   3.5 Related Work  . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
Acknowledgments

Thank you
Programming systems that include mechanical verification as part of the compilation process will make programs more efficient, reliable, and flexible - if they can ever be made practical.

Techniques for Program Verification
GREG NELSON
CHAPTER 1

Introduction

By translating each program statement into its mathematical equivalent and conjoining the intended program outcomes, one can formulate a formal logical representation of the program. Together with some background theories such as integer arithmetics and theory of array, one can reason about the correctness of constructed model—a proof procedure on programs more commonly known as deductive program verification.

With the ever increasing ubiquity of programs controlling more and more aspects of our lives, computer automation has already became an indispensable complexity running our power grids, cars, and global finance. Correctly implemented program automation alleviates us from tedium and reduces superfluous labour costs. Yet the erroneous ones, often known as bugs, can be lurking among the thousands of lines of deployed code, waiting for the condition which has never been conceived before to trigger havoc on our phones, bank accounts, or aeroplanes.

If "testing can never show the absence of bugs" [18], the only way to assure correctness is through formal verification. Although program verification started back in Turing’s time [?], the practice of deductive verification is still considered time-consuming and requires highly trained experts. To convince the industry, automated program verifier is the crucial step forward. More importantly, how do we provide an automated program verifier that is batteries-included without the need of shipping an expert along?

Leveraging on one of the popular intermediate verification languages, Boogie [42], this thesis explores alternative logic solvers to reduce user annotation effort and to improve flexibility. Moreover, this thesis further investigates volatile behaviors commonly found in automated program verifiers. Through our proposed automatic testing technique, developers of automated program verifiers can systematically identify the so-called butterfly effects [46] and improve verifiers’ robustness.

In the remaining chapter I briefly summarize existing challenges Sec. 1.1 and state-of-the-art Sec. 1.2. Specific contributions Sec. 1.3 are listed before a short conclusion with future research direction. Chapter 2 and 3 each encloses one
1.1 Grand Challenge: Practical Program Verification

Software verification can ensure both safety and functional correctness, but the practice of program verification is often considered “developed by experts for the experts” [?] thus prohibitively expensive. Application of formal verification is still limited to critical software components in industry. It is important to stress that formal verification can and have been done for large projects [?]. With interactive proof assistants such as Coq [] and Isabelle/HOL [] one can obtain correctness proof for arbitrary programs. Despite recent improvements in automatic tactics, interactive approach is still heavily dependent on highly trained proof engineers.

To make program verification practical, the verification tools must be as automatic as possible. The surge of automated program verifiers together with reusable infrastructures in the past decade has lowered the complexity of applying program verification. "Push-button" logic solvers can now discharge constraint problems in splits of a second and widely used in different domains such as circuit design, planning, and control optimization [?]. Integrated specification languages such as Java Modeling Language (JML) [?] enables programming to seamlessly argument programs with specification.

Still, making program verification practical for serious users remains one of the grand challenge. According to Leino and Moskal [?], critical issues hindering usability include: 1) difficulties in controlling automated logic solvers 2) cryptic error messages from proof failures 3) prolonged waiting loops 4) additional learning curve for the uninitiated. Despite aforementioned breakthroughs, these issues remain user frustration and making formal verification inaccessibly expensive.

Assuming we want to verify a small program: this program inputs an array of integers and check if the integers are arranged in an increasing order, how would one express this intended property? In other words, how to express sortedness as a program property for verification tools. One intuitive formulation states that every element should be less than or equal to its immediate neighbour:

\[ \text{element}_i \leq \text{element}_{i+1} \] (1.1)

A quick observation suggests boundary conditions to guard against indexing out of bounds—an array-specific knowledge required during reasoning process—
together with common syntactic features of arrays:

\[ \forall i, 0 \leq i < a.\text{length} - 1 \implies a[i] \leq a[i+1] \quad (1.2) \]

In fact, even with state-of-the-art verification systems, property \texttt{Equation 1.2} can still cause hiccups during automated verification attempts. A verification veteran would suggest the following equivalent for smoother automated verification experience:

\[ \forall i, j, 0 \leq i < j < a.\text{length} \implies a[i] \leq a[j] \quad (1.3) \]

And indeed property \texttt{Equation 1.3} can be easily discharged by the same prover in fraction of a second. This problem is at the heart of present thesis: \textit{how to make automated program verification push-button without further frustrations?} In other words, we aim to explore a more flexible and robust automated verification.

In the following section \texttt{Sec. 1.2}, we describe state-of-the-art automated program verification techniques and introduce intermediate verification languages (IVLs). Our contributions are described in section \texttt{??} followed by a conclusion and our future research plan.

1.2 State of the Art

Deductive program verification essentially translates target program together with the correctness properties—safety and/or functional—into mathematical statement. Together with background theories involved, the statement can be discharged by any reasoning engine ranging from interactive proof assistants to automated theorem provers, even by pen and paper proofs [?]. Final result of deductive verification consists of sequences of deductive steps, suggesting that the program indeed adheres to its specification without any exception. Unlike theorem proving in a mathematical context, deductive program verification often requires tedious and dull routines which should be mechanized. The focus of this thesis is to explore possible improvements on automated deductive verification. Let us first look at one crucial engineering advent fostering automated program verification: Intermediate Verification Language.

**Intermediate Verification Language.** Although it is possible to directly encode production-level programming languages into logic statements, this approach prevents modular reuse of verification components. Deciding how to logically model each aspect of a programming language, such as memory and heap, while maintaining verification efficiency is an artful process. The challenge of handling hundreds of different programming languages, each with their
own peculiarities, is difficult but nothing new to the field of computer science. Similar to intermediate representation (IR) in compiler technology, one can tackle the aforementioned challenge with Intermediate Verification Languages (IVLs).

IVLs are designed to interface various programming constructs with lower-level logical formalism and often designed to be a small core language. The goal of a successful IVL is not to encode all sorts of sophisticated abstractions with minimum syntax but to provide a set of primitives for modeling different abstractions. With a core intermediate language, the development of program verifier can be split into two parts: first encode front-end programming languages using IVL followed by generating verification condition from the corresponding IVL program. The verification condition, often abbreviated as VC, entails correctness properties of original program and can be generated using well-known calculus such as weakest precondition [?]. Another benefit of employing intermediate language is that the underlying VC can be store in a human-readable format and can be dispatched to different reasoning engines if needed. There are two most popular intermediate verification languages at the time of writing: Boogie [42] developed by Leino et al. and Why3 [22] developed by Filliâtre et al.

Boogie, sitting at the heart of formal verification at Microsoft Research, is an IVL developed under close integration with one of best solver Z3 []. With many diverse front-ends [43], Boogie is one of the most popular IVL. Boogie refers to both the intermediate language (previously known as BoogieIVL) and the verification condition generator that outputs SMT queries. Boogie the language consists of pure declarations such as axioms and types as well as imperative procedures. Common imperative features such as conditional, loops, jumps, and procedure calls are all included as Boogie primitives. Despite designed to function as intermediate language, user can also directly write/modify Boogie programs with fine-grain control over specification.

Why3 is an IVL platform designed with modular back-ends in mind. Originally developed as a tactic of Coq, the verification tool Why was one of the first verification tool implemented to accommodate multiple back-end solvers [?]. The latest iteration, Why3 [?], combines a library of logic theories written in its ML-like intermediate language called WhyML with translation schemata to facilitate multiple back-end solvers. Solvers are typically designed with a particular target domain of logic problems and often compliment other solvers. Why3 provides configurable tools to convert the specification into various formats, each tailored to different back-end provers. Due to its ML-like language characteristic, Why3 is especially suitable to model functional languages.

\footnote{The tactic was called Correctness}
Alternatively, one can design specialized specification language for best reasoning efficiency on target language. For example, KeY uses modal logic\(^2\) to model Java with JML specification. Viper \([\) builds atop Boogie and provide extensions on permission logic, well suited for reasoning concurrent program properties. SeaHorn \([\) tackles huge collection of front-end language by interfacing the popular compiler IR—LLVM \([\)—and internally encodes its VCs as constrained Horn clauses (CHC).

In this thesis, we focus our contributions on one of the popular IVL Boogie to make our work immediately reusable by others. However, the principle ideas are not limited to any particularities of Boogie and therefore applicable to any other IVLs.

**Automated Logic Solvers.** Generally speaking, logic solvers are computer programs designed to aid human in logic reasoning. In mathematics, the goal of logic reasoning can range from proving four color theorem to proving Kepler conjecture \([\). In computer science, the application of a logic solver can go from model checking, type inference, to circuit synthesis. Many developments, both in hardware and software, benefit from using logic solver as a black-box reasoning engine. The underlying complexity is to decide the trade-off between expressive formalism and solver’s automation efficiency. Starting at propositional logic, despite the proven NP-complete complexity, Boolean satisfiability (SAT) solvers are highly automated, and thanks to recent advents such as conflict-driven learning SAT problems can be solved fairly efficiently. At the other end of expressiveness spectrum lies the higher-order logic formalism. With such embedded expressiveness, higher-order problems tend to require extensive manual guidance throughout the proof in order to avoid divergence. Most prominent examples include interactive proof assistants Isabelle/HOL and Coq.

Looking at verification condition of programs, a natural choice of logic formalism is the first-order logic (FOL). Equipped with quantification operators (\(\forall, \exists\), first-order logic can be used to encode program properties without exponential growth on notation. Combined with theory of equality, first-order logic can express rich theories to support reasoning in program verification. By lifting the satisfiabilty algorithm, DPLL(T) \([\] seamlessly integrates various theory decision procedures into Satisfiability Modulo Theory (SMT) solvers. In principle, SMT solver can only deal with decision procedures, hence excluding quantified reasoning. But with the work \([\)…

Aside from SMT-style of solvers, there exists another approach of reasoning first-order logic. Based on logic resolution and careful treatments on equality

\(^2\)More precisely, KeY uses a specialized extension called Java Card Dynamic Logic to represent proof obligations
1.3 Contributions

This thesis presents contributions to the field of intermediate verification in the form of two research publications. Both of these contributions are accompanied by open source implementations based on BoogieIVL [42] and available at:

http://www.cse.chalmers.se/~yutingc/


Quantifiers (∀, ∃) are inherently difficult for SMT solvers due to need of explicit variable instantiation. This issue is traditionally resolved using user-provided heuristic, which guides the SMT solver during its proof search, known as the triggers. Triggers are non-trivial annotations requiring in-depth expertise and are solver dependent. In the paper reproduced in this thesis, we present an alternative encoding which enables the use of resolution based backend solvers for better quantifier reasoning.

- Using its recent extensions, our tool inputs Boogie program and generate verification condition in TPTP. User can now use alternative back-ends such as first-order theorem provers to discharge proof obligations of Boogie.

- This approach alleviates the need of manual annotation for quantifiers completely.

- Based on our tool, we conclude a significant improvement on quantifier reasoning.

The paper has been accepted and published in the peer-reviewed 13th International Conference on integrated Formal Methods (iFM 2017) and presented in Torino, Italy.

[Paper 2] Robustness Testing of Intermediate Verifiers

Automated solvers are known to be highly volatile with respect to its input. This volatility hinders the reliability of intermediate verification tier and often requires the user to be familiar with the idiosyncrasies. Consequently, it is often difficult for naive user to distinguish between the brittle behaviour of a verifier from the actual proof failure. In this paper we present an automatic technique to
detect the brittleness of program verifiers. This technique, for which we called robustness testing, is fully automatic and lightweight. By introducing different semantic preserving mutations, our tool and technique is able to explore volatile behaviours of underlying solvers.

- We present an automatic lightweight testing tool aiming for robustness testing of Boogie verifier platform.
- Our tool generates semantic equivalents of input Boogie program. These mutants can be used for benchmark purposes.
- Our experiment result shows significant percentage of syntactical mutants breaking verifiability hence identifying volatility of solver.
- Syntactical mutation operators can be easily specified and our tool provides adjustable interface to configure each experiments.

This paper has been accepted for publication in the peer-reviewed 16th International Symposium on Automated Technology for Verification and Analysis (ATVA 2018) and will be presented in Los Angeles, USA.

1.4 Conclusion and Future Work

Based on intermediate verification language Boogie, this thesis aims at improving practicality of automated program verifiers. Boogie less triggers (BLT) presents alternative encoding to bridge existing Boogie programs with resolution-based theorem prover, alleviating manual annotation burden. Robustness testing presents a testing technique to better identify existing fragile behaviors of automated verifiers, aiding the development of more stable verifiers.

**Real Push-Button Verification.** One major objection for formal methods is the cost of specification. Writing specification is indeed expensive but such investment can be justified by future reuse: functionality of a component remains stable while its implementation can be later refined. There are also research efforts on specification generation—such as loop invariants []—to aid specification engineers. However, additional annotation—such as matching patterns for quantifiers—hamper verification efforts and further introduce instability to verification tools. Reducing dependency on such annotations can improve predictability of verification tools, making the learning process more accessible. The flexibility introduced by our exploratory tool BLT originates from the desire to reduce such annotation effort. Power user can still opt for surgical annotation to guide the underlying solver while non-expert user can obtain correctness
proof via alternative solver at the cost of overall efficiency. We plan to further de-
velop and mature BLT to accommodate complete Boogie language and integrate
multi-solver Boogie back-end.

Verifying the Verifier: Bootstrap Verification Tools. Verification tools need verification too. The ultimate assurance is to bootstrap verification tools themselves, but this can be challenging even for interactive proof assistants. Nevertheless, a predictable and stable verification system is still expected in order to become practical. To mitigate the mysterious butterfly-effects, one must first identify under which syntactic variant will the prover encounter hick-ups. The robustness testing we proposed can be a light-weight automatic approach to unveil possible causes of aforementioned fragility. This tool can be further adapted for robustness testing at different level of encoding such as SMT-Lib in order to pinpoint the root cause. Our future plan includes reduction on failing mutants to provide more precise trace of fragile behavior.

Proving program correctness is difficult. Creating a generic automation for all programs is even more difficult. Despite the wonderful success of SMT solvers in the past decade, quantified reasoning will always remain a huge chal-
lenge for SMT solvers. The SMT-Lib (set-logic ALL) option essentially boils down to tactic suites encoding different instantiation order for quantifiers. In fact, despite created the most popular SMT solver Z3 himself, de Moura [?] has been recently proposing White-box automation based on his new theorem prover called Lean. By allowing user to “roll-your-own-tactics”, Lean tackles aforementioned challenge with extra layers of interactivity and control over au-
tomated prover. The idea of “interactable” SMT solver is interesting but can require extra user expertise.

Looking back at the critical issues [?], we have seen dramatic improve-
ments on solver efficiency, leading to much rapid feedback loops for verification users. With intermediate verification languages as stepping stone into pure logic, beginners can now learn to reason at the intermediate level or just leave out the encoding process to verification tool developers. Still, automated verification tools suffer from cryptic error message, unstable proof, and difficulties in controlling automated logic provers. Proof stability is crucial, both for tool developers and users. While existing verification tools are gaining attention as a mean of software quality assurance, it is important to start thinking about maintainability of specification and correctness proof along software life cycle. Finally, the fine balance between full automation and user control is the grand challenge. One promising approach tackling this issue is to shepherd multiple back-end provers [?] and their complementary specializations to optimize overall automation. Another approach is the "auto-active” approach [].

10
CHAPTER 2

Triggerless Happy: Intermediate Verification with a First-Order Prover

Yu-Ting Chen, Carlo A. Furia

Abstract. SMT solvers have become de rigueur in deductive verification to automatically prove the validity of verification conditions. While these solvers provide an effective support for theories—such as arithmetic—that feature strongly in program verification, they tend to be more limited in dealing with first-order quantification, for which they have to rely on special annotations—known as triggers—to guide the instantiation of quantifiers. Writing effective triggers is necessary to achieve satisfactory performance with SMT solvers, but remains a tricky endeavor—beyond the purview of non-highly trained experts.

In this paper, we experiment with the idea of using first-order provers instead of SMT solvers to prove the validity of verification conditions. First-order provers offer a native support for unrestricted quantification, but have been traditionally limited in theory reasoning. By leveraging some recent extensions to narrow this gap in the Vampire first-order prover, we describe a first-order encoding of verification conditions of programs written in the Boogie intermediate verification language. Experiments with a prototype implementation on a variety of Boogie programs suggest that first-order provers can help achieve more flexible and robust performance in program verification, while avoiding the pitfalls of having to manually guide instantiations by means of triggers.
2.1 The Trouble with Triggers

Deductive verification reduces the problem of assessing the correctness of a program to checking the validity of logic formulas known as verification conditions (VCs). VCs normally include both first-order quantification and theory-specific fragments: quantifiers naturally express specification properties of the program under verification—such as its heap-based memory model, or an inductive definition of “sortedness”; logic theories, on the other hand, are needed to reason efficiently about basic data types—most notably, integers. Having both kinds of logic in the same formulas aggravates the already challenging problem of automated reasoning.

SMT solvers are the tools of choice to check the validity of VCs, and in this role they are part of nearly every verification toolchain. Such solvers expressly target combinations of decidable logic theories (the “T” in SMT is for “theory”) on which they achieve a high degree of automation; in contrast, they tend to struggle with handling the complex usages of quantification that are often necessary for expressing VCs but render logic undecidable. The practical solution that has been adopted in most SMT solvers is to use triggers [17]—heuristics that guide the instantiation of quantifiers. Triggers are specific to the axioms that define the predicates used in a formal specification; as such, they are additional annotations that must be provided for verification. Writing triggers that achieve good, predictable performance remains a highly specialized skill—a bit of a black art that only few researchers are fluent in.

In contrast to SMT solvers, first-order theorem provers support, as the name suggests, first-order quantification natively and without particular restrictions. First-order provers have not been often used in program verification for a number of reasons, including the more spectacular performance improvements of SAT/SMT solvers, and the lack of out-of-the-box support for theory-specific reasoning. More recently, however, these limitations have started to mollify, and the best first-order provers have become flexible tools with some effective support for arithmetic and other commonly used theories. Encouraged by these improvements, in this paper we probe the feasibility of using first-order provers in lieu of SMT solvers to check the validity of VCs for the deductive verification of programs.

To make our contributions applicable to the verification of a variety of programming languages, we target the popular intermediate verification language Boogie—which we outline in the motivating examples of Sec. 2.2. Boogie is both a language and a tool: Boogie the language combines an expressive typed logic and a simple imperative procedural programming language, and Boogie the...
tool generates VCs from Boogie programs in a form suitable for SMT solvers; the Boogie language also includes syntax for triggers, which are passed on to the back-end solver to help handle quantifications.

We developed a technique and a tool called BLT (Boogie less triggers), which inputs Boogie programs and generates VCs in a subset of the TPTP (Thousands of Problems for Theorem Provers) format \[58\] that is suitable for first-order provers. In Sec. 2.3 we describe the salient features of the first-order encoding, and the key challenges we addressed to produce VCs that are tractable. To this extent, we specifically took advantage of some recent features of TPTP supported by the Vampire prover \[37\] \[35\] to encode imperative code effectively. Based on experiments involving 126 Boogie programs, in Sec. 3.4 we demonstrate how BLT can achieve better stability and flexibility in a variety of situations that depend on triggers when analyzed using the SMT solver Z3 (Boogie’s default back-end solver).

The main advantage of using a first-order prover is that complex quantifications are handled by the prover without requiring trigger annotations—thus helping increase the degree of automation, and reduce the expertise required to use verification technology proficiently. In Sec. 2.6 we discuss some outstanding challenges of improving the flexibility of deductive verification that we intend to address to extend the present paper’s work in this direction.

In the paper, “Boogie” refers to the behavior of the Boogie tool with its standard back-end Z3, whereas “BLT” refers to the behavior of the BLT tool, which also inputs Boogie programs but feeds VCs to the Vampire first-order prover. To simplify the presentation, we often attribute to Boogie qualities that more properly belong to Boogie used in combination with Z3—namely, the effect of triggers.

**Tool availability.** The tool BLT and the examples used in the paper are available as open source at:

https://emptylambda.github.io/BLT/

### 2.2 Motivating Examples

This section discusses examples of programs where the outcome of verification using Boogie (with Z3 as back-end solver) crucially depends on triggers; BLT, which generates VCs for the Vampire first-order prover, is not affected by triggers, and thus behaves in a more predictable and robust way on such examples. Sec. 3.4 discusses a more extensive experimental evaluation.

**Matching triggers.** Boogie dispatches VCs to an SMT solver, which may need help to decide how to instantiate universally quantified variables while
% type ref;
const nil: ref;
const next: [ref] ref;

function dist(from, to: ref) returns (int);
axiom (∀ from, to: ref • (dist(from, to) = 0) \land
(from ≠ to \implies dist(from, next[to]) = dist(from, to) + 1));

procedure length(head: ref) returns (len: int)
ensures len = dist(head, nil);
{ cur, len := head, 0;
while (cur ≠ nil)
  free invariant head ≠ cur;
  invariant len = dist(head, cur);
  { cur, len := next[cur], len + 1; }
}

% const a: [int] int;

axiom (∀ i: int • // a is sorted
0 ≤ i \implies a[i] < a[i+1]);

function hash(int) returns (int);
axiom (∀ x, y: int • x > y \implies hash(x) > y);

procedure ah(k: int) returns (h: int)
requires k ≥ 0;
ensures h > a[k];
{ h := hash(a[k+1]); }

(b) Reasoning about hash functions.

(a) Length of a linked list.

Figure 2.1. (a): trigger ⟨dist(from, to)⟩ in the axiomatic definition of function dist is required to prove that the loop invariant in procedure length holds initially. (b): axiomatic definitions of sortedness and of hashing are ineffective in proofs even if they are semantically sufficient to verify procedure ah.

searching for a proof. A trigger (also called matching pattern) is a directive to the SMT solver on how to instantiate quantifiers to create new terms based on the terms that are already in the proof space. A trigger ⟨f(x)⟩, associated with a universally quantified variable x, instructs the SMT solver to instantiate x with the value E whenever the ground term f(E) is in the proof state. Picking suitable triggers is not trivial, as it risks introducing problems in opposite directions: triggers that are too permissive generate otiose terms that may slow down a proof, or even set off an infinite loop of term generation; triggers that are too specific miss terms that are necessary for a proof, and thus ultimately reduce the level of proof automation. To make things even more complicated, SMT solvers introduce their own default triggers when no user-supplied triggers are available, which renders the whole business of understanding and selecting triggers a mighty tricky one.

Linked lists. In the example of Fig. 2.1a inspired by one of Boogie’s online examples, next is a map from nodes of type ref to their successors in a chain of linked nodes—a straightforward model of a heap-allocated linked list. Function dist defines that the distance between two nodes from and to is the number of hops following next from one node to the other. Procedure length computes such distance with a simple loop that starts from a given node head and follows next until it reaches a nil node—indicating the end of the list. If the list is

http://www.rise4fun.com/Boogie/5I
acyclic—an assumption we encode with the invariant head $\neq$ cur (declared as free, and thus assumed without checking it)—length satisfies its specification that it returns the value $\text{dist}(\text{head}, \text{nil})$. Still, without trigger $\langle \text{dist}(\text{from}, \text{to}) \rangle$, Boogie fails to verify the procedure; precisely, it cannot prove that the loop invariant holds initially—that is, that $0 = \text{dist}(\text{head}, \text{head})$—even if this is a mere application of the base case in $\text{dist}$’s axiomatic definition.

For a successful correctness proof, Boogie requires either that the axiom defining $\text{dist}$ be split into two axioms—one for the base case and one for the inductive case—or that the trigger $\langle \text{dist}(\text{from}, \text{to}) \rangle$ be added to $\text{dist}$’s definition. Even this simple example indicates that predicting the behavior of quantifier instantiation, and the need for triggers, imposes an additional burden to users, and renders the verification process less robust. In contrast, BLT verifies the very same example without any user-provided suggestions about how to instantiate quantifiers, and without depending on the axioms being in a specific form.

**Hash functions and sortedness.** Using quantified formulas with SMT solvers often leads to brittle behavior: changes to a formula that do not affect its semantics may make it significantly less effective in proofs. Take the example of Fig. 2.1b where map $a$ models an unbounded integer array whose elements are sorted in strictly increasing order. Function hash has the property that the hash of an integer $x$ is greater than any integer smaller than $x$. By combining these two properties, it should be possible to verify procedure $\text{ah}$, which inputs a nonnegative integer $k$ and returns the hash of $a[k+1]$—which has to be greater than $a[k]$. Boogie, however, fails verification of $\text{ah}$’s postcondition.

In an attempt to help the SMT solver, we may try to add triggers to the axioms in the example. However, we cannot add triggers to the axiom about hash: in order to be sufficiently discriminating [17][3], a trigger must mention all quantified variables ($x$ and $y$ in this case), cannot use theory-specific interpreted symbols (such as $\langle x > y \rangle$), because matching does not know about function symbols interpreted by some theory, and cannot mention variables by themselves (such as $\langle x, y \rangle$), because a variable by itself would match any ground term. Since $y$ only appears by itself or in arithmetic predicates, no valid user-provided trigger involving $y$ can be written. What about adding triggers to the axiom that declares a sorted? Here the only sensible trigger is $a[i]$, which however results in a matching loop: an infinite chain of instantiations that quickly saturate the proof space.

As observed elsewhere [45][44] and part of the folklore, an equivalent definition of sortedness that works much better with SMT solvers uses two quantified
Boogie can verify \( a_h \) if we use the definition of sortedness in (2.1) instead of the one in Fig. 2.1b. Somewhat surprisingly, Boogie can also verify \( a_h \) if we use the same definition as in Fig. 2.1b but we add it as a \textit{precondition} to \( a_h \) rather than as an axiom. In contrast, BLT easily verifies any of these semantically equivalent variants: while first-order theorem provers are not immune from generating infinite fruitless instantiations, their behavior does not incur the brittleness that derives from depending on suitable triggers—that are neither too permissive nor too constraining.

2.3 Encoding Boogie in TPTP

In order to use first-order provers to verify Boogie programs, we define a semantic-preserving translation \( T \) of the Boogie language into TPTP—the standard input format of first-order theorem provers.

As a result of continuous evolution, TPTP has become a sizable language that aggregates several different logic fragments, going well beyond classic first-order predicate calculus. We loosely use the name TPTP to refer to the specific subset targeted by our translation, which mainly consists of a monomorphic many-sorted first-order logic, augmented with the so-called FOOL fragment: a first-class Boolean sort and polymorphic arrays. Our translation is informed by the recent support for FOOL \cite{37} added to the Vampire automated theorem prover, so that we can use it in our experiments as an effective back-end to verify Boogie programs.

Boogie combines a typed logic and a simple imperative programming language; Sec. 2.3.1 discusses the translation of the former, and Sec. 2.3.2 the translation of the latter. We outline the essential features of Boogie and TPTP as we describe the translation \( T \).

2.3.1 Declarative constructs

\textit{Types}. Boogie’s \textit{primitive types} include \texttt{int} (mathematical integers) and \texttt{bool} (Booleans), which naturally translate to TPTP’s integer type \$\text{int}\$ and Boolean type \$\text{o}\$. Vampire reasons about terms of type \$\text{int}\$ using an incomplete first-order axiomatization of Presburger arithmetic, sufficient to handle common usages in program analysis.

A Boogie \textit{user-defined type} declaration \texttt{type t} introduces an uninterpreted
type $t$ expressed in TPTP by a type entity $t$ of type $\texttt{tType}$, which represents
the type of all primitive uninterpreted types.

A Boogie map type $[t_1, \ldots, t_n]$ $u$ corresponds to a mapping $t_1 \times \cdots \times t_n \rightarrow u$, which translates to a curried array type $\mathcal{T}(t_1) \rightarrow \cdots \rightarrow \mathcal{T}(t_n) \rightarrow \mathcal{T}(u)$ in TPTP:

$$\mathcal{T}([t_1, \ldots, t_n] u) = \begin{cases} \texttt{array}(\mathcal{T}(t_n), \mathcal{T}(u)) & n = 1 \\ \mathcal{T}([t_1][[t_2, \ldots, t_n]u) & n > 1 \end{cases}$$

We currently do not support other Boogie types—notably, reals, bitvectors, and polymorphic types and type constructors.

**Declarations.** TPTP declarations are expressions of the form $\ell(I, K, D)$, where $\ell$ denotes a specific subset of TPTP, $I$ is an identifier of the declaration, $K$ is the kind of declaration (type, axiom, or conjecture), and $D$ is the actual declaration. Here we simply write `tptp` for $\ell$ and omit the identifier $I$—which is not used anyway. Then, a constant declaration `const c: t` in Boogie translates to the TPTP declaration `tptp(type, c: T(t))`. An axiom `axiom ax` in Boogie translates to a TPTP axiom `tptp(axiom, \mathcal{T}(\text{ax})`). Sec. 2.3.2 describes other kinds of declarations, used to translate imperative constructs.

**Functions.** Mathematical functions are part of both Boogie and TPTP; thus the translation is straightforward: function declarations translate to function declarations

$$\mathcal{T}(\text{function } f(a_1: t_1, \ldots, a_n: t_n) \text{ returns } (u)) = \text{tptp(type, } f: (\mathcal{T}(t_1) \circ \cdots \circ \mathcal{T}(t_n)) \mapsto \mathcal{T}(u))$$

and function definitions are axiomatized.

**Expressions.** Boolean connectives translate one-to-one from Boogie to TPTP. The integer operators $+$ and $-$ translate to built-in binary functions $\texttt{sum}$ and $\texttt{difference}$; similarly, integer comparison uses built-in functions such as $\texttt{less}$ and $\texttt{greatereq}$, with obvious meaning. The equality and non-equality symbols have the same meaning in Boogie and in TPTP: $x = y$ iff $x$ and $y$ have the same type and the same value.

Boogie map expressions translate to nested applications of TPTP’s $\texttt{select}$
and \$store, which behave according to the axiomatization of FOOL \cite{37}:

\[
\begin{align*}
\tau(m[e_1, \ldots, e_n]) &= \begin{cases} 
\$select(\tau(m), \tau(e_n)) & n = 1 \\
\tau((m[e_1])[e_2, \ldots, e_n]) & n > 1 
\end{cases} \\
\tau(m[e_1, \ldots, e_n := e]) &= \begin{cases} 
\$store(\tau(m), \tau(e_n), \tau(e)) & n = 1 \\
\tau(m[e_1 := (m[e_1])[e_2, \ldots, e_n := e]) & n > 1 
\end{cases}
\end{align*}
\]

where \(m\) is an entity of type \([t_1, \ldots, t_n] \cup\).

Quantifiers. Quantified logic variables must have identifiers starting with an uppercase letter in TPTP, and thus \(\tau\) may rename logic variables. As we repeatedly mentioned, triggers (associated with quantifiers) have no use in TPTP and thus the translation drops them. Note that the fragment of TPTP targeted by our translation also supports quantification over variables of array type; thus, we end up with a form of second-order quantification—which is restricted by the peculiarities of the FOOL encoding \cite{37} but is sufficient to deal with the most common usages of maps in Boogie.

### 2.3.2 Imperative constructs

**Variables.** Program variables encode state, which is modified by computations. In the logic representation, a Boogie program variable \texttt{var v: t} translates to the TPTP declaration \texttt{tptp(type, v: T(t))}, which corresponds to a free logic variable of given type. Indeed, constants and program variables have the same TPTP representation, with VCs encoding the effects of computations in a purely declarative way.

**Procedures.** Boogie’s imperative constructs define procedures, each consisting of a signature, a specification, and an implementation, as shown in Fig. 2.2a. Each procedure determines a set of VCs that encode the correctness of the procedure’s implementation against its specification.

Fig. 2.2b shows the TPTP translation \(\tau(p)\) of \(p\), which consists of three parts:

1. The input/output arguments of \(p\), which are encoded as if they were global program variables; since each procedure is translated independent of the others, there is no risk of interference.

2. The precondition of \(p\) (\texttt{requires R}), which is encoded as an axiom.

3. The actual VCs of \(p\), which are encoded as a TPTP conjecture expressing that the implementation B determines a sequence of states that end in a state satisfying \(p\)’s postcondition (\texttt{ensures E}).
In the rest of this section, we define the predicate transformer \( \tau(S, Q) \), which behaves like a weakest precondition calculation \[28\] of predicate \( Q \) through Boogie statement \( S \).

If a theorem prover can prove the conjecture from the given axioms, the implementation of \( p \) is (partially) correct against its specification.\[3\] Fig. 2.3b shows the complete translation of the example in Fig. 2.1b including functions, axioms, arrays, and assignments.

\[
\begin{align*}
\% & \quad \text{// signature} \\
\text{procedure} & \quad p(a_1 : t_1, \ldots, a_n : t_n) \\
\text{returns} & \quad (b : u) \\
\% & \quad \text{// specification:} \\
\text{requires} & \quad R \\
\text{modifies} & \quad M \\
\text{ensures} & \quad E \\
\% & \quad \text{// implementation:} \\
\{} & \quad B \\
\\end{align*}
\]

(a) Generic Boogie procedure \( p \).

(b) Encoding of \( p \)'s VCs in TPTP.

Figure 2.2. General structure for the translation of a Boogie procedure.

**Sequential composition.** The encoding of statements is naturally compositional:

\[
\tau(S ; T, Q) = \tau(S, \tau(T, Q))
\]

**Assignments.** The encoding of assignments uses the *let-in* construct:

\[
\tau(v := e, Q) = \text{$let$} (\tau(v) \triangleq \tau(e), Q)
\]

which roughly corresponds to introducing a fresh variable \( v' \), defining its value according to \( \tau(e) \), and replacing every free occurrence of \( \tau(v) \) in \( Q \) by \( v' \).

The encoding of nondeterministic assignments (“havoc”) uses the derived scheme \( \tau(\text{havoc} \ v, Q) = \tau(v := v', Q) \), where \( v' \) is a locally fresh variable—introduced by the translation—of the same type as \( v \) without other constraints on its value.

**Passive statements.** The encoding of assertions and assumptions follows the standard weakest precondition rules:

\[
\begin{align*}
\tau(\text{assert} \ b, Q) & = \tau(b) \land Q \\
\tau(\text{assume} \ b, Q) & = \tau(b) \implies Q
\end{align*}
\]

**Procedure calls.** A call \( \text{call} \ r := p(e_1, \ldots, e_n) \) to procedure \( p \) in Fig. 2.2a desugars the call using standard *modular verification semantics*, where the

\[3\] The typechecker establishes the correctness of a procedure’s *modifies* clause, so that the prover can just rely on it. This is possible because Boogie’s variables cannot be aliased.
callees effects within the caller are limited to what the callees specification declares:

\[ \tau(\text{call } r := p(e_1, \ldots, e_n), Q) = \tau(\text{assert } R(e_1, \ldots, e_n); \text{ havoc } r, M; \text{ assume } E(e_1, \ldots, e_n, r), Q) \]

**Loops.** Encoding a loop \( \tau(\text{while } (b) \text{ invariant } J \{ L \}, Q) \) involves three logically conjoined conditions:

1. **Initiation** checks that the invariant holds upon entering the loop: \( T(J) \).

2. **Consecution** checks that the invariant is maintained by the loop:
   \[
   \tau(\text{havoc } \theta(L); \text{ assume } b \land J; L, T(J)) \quad \text{where } \theta(L) \text{ are the targets of the loop body—variables that may be modified by } L; \text{ these are just the variables that appear as targets of assignments, as arguments of havoc statements, or in the modifies clauses of procedures called in } L.
   \]

3. **Closing** checks that the invariant establishes \( Q \) (the loops postcondition):
   \[
   \tau(\text{havoc } \theta(L); \text{ assume } \neg b \land J, Q).
   \]

The tool \texttt{BLT} generates a TPTP conjecture for each of these conditions, which are proved independently; thus, in case of failed verification, we know which VC failed verification. Fig. 2.3a shows the VC corresponding to closing of procedure length’s loop from Fig. 2.1a.

**Abrupt termination.** Statements such as \texttt{goto} and \texttt{return} make imperative code less structured, and complicate the encoding of VCs. We currently do not support \texttt{goto} and \texttt{break}, whereas we handle \texttt{return} statements: for every simple path \( \pi \) on the control-flow graph of the procedure \( p \) being translated that goes from \( p \)'s entry to a \texttt{return} statement, we generate the additional VC \( \tau(\tilde{\pi}, T(E)) \)—where \( E \) is \( p \)'s postcondition and \( \tilde{\pi} \) is the sequence of statements on \( \pi \), suitably modified to account for conditional branches and loops. For brevity we omit the uninteresting details.

**Conditionals.** TPTP includes the conditional expression \$ite(b, \text{ then }, \text{ else})\—which evaluates to \text{ then} if \( b \) evaluates to true, and to \text{ else} otherwise. Using \$ite and first-order Booleans, we could encode the VC for a Boogie conditional statement as:

\[
\tau(\text{if } (b) \text{ then } \{ \text{ Th } \} \text{ else } \{ \text{ El } \}, Q) = \text{site}(T(b), \tau(\text{Th}, Q), \tau(\text{El}, Q))
\]

(2.3)

As noted elsewhere [41], (2.3) tends to be inefficient because it duplicates formula \( Q \), so that the generated VC is worst-case exponential in the size of the
% % fresh variables cur_ and len_.
tptp(type, cur_: ref).
tptp(type, len_: $int).
% VC checking "closing" of the loop
% initial assignments
cur ≜ head; len ≜ 0,
% generic number of loop iterations
$let([cur, len] ≜ [cur_, len_],
% assume exit condition
(¬(cur ̸= nil) ∧
% assume invariant
(len = dist(head, cur)) ∧ (head ̸= cur)))
% assert postcondition
⇒ (len = dist(head, nil))).

(a) TPTP translation of one VC of Fig. 2.1a.

% % array a
% a is sorted
% function hash
$tptp(type, a: $array($int, $int)).
$tptp(axiom, ∀[I: $int]: ($less($select(a, I), $select(a, $sum(I, 1))))).
$tptp(type, hash: $int⇒ $int).
$tptp(axiom, ∀[X: $int, Y: $int]:
($greater(X, Y) \implies
$greater(hash(X), Y))).
$tptp(axiom, ∀[X: $int, Y: $int]:
($greater(X, Y) \implies
$greater(hash(X), Y))).
$tptp(axiom, ∀[X: $int, Y: $int]:
($greater(X, Y) \implies
$greater(hash(X), Y))).
$tptp(conjecture, $let(
% initial assignments
h ≜ hash($select(a, $sum(k, 1))),
% assume exit condition
(¬(h ̸= nil) ∧
% assume invariant
(len = dist(head, cur)) ∧ (head ̸= cur)))
% assert postcondition
⇒ (len = dist(head, nil))).

(b) TPTP translation of Fig. 2.1b.

Figure 2.3. Excerpts of BLT’s TPTP encoding of the examples in Fig. 2.1.

input program.

Instead of following [23, 41]’s approach, based on passivization, we leverage another feature of FOOL, namely *tuples*, to build a VC whose size does not blow up. A code block is *purely active* if every statement it contains is an assignment or a conditional whose branches are purely active. Given a purely active code block $B$, $lhs(B)$ denotes the variables assigned to anywhere in $B$. Given a purely active conditional statement, we encode it using TPTP tuples and conditional expressions as:

$$\tau(\text{if} \ b \ \text{then} \ \{ \ Th \} \ \text{else} \ \{ \ El \}, Q) =$$
$$\$let(\[lhs(Th)\]⊕[lhs(El)]) \triangleq \$ite(T(b), \tau(Th, [lhs(Th)]), \tau(El, [lhs(El)]), Q)$$

(2.4)

Operator $\oplus$ denotes a kind of tuple concatenation where variables that appear in both tuples only appear once in the concatenation; for example $[x, y, z] \oplus [x, w, z] = [x, y, z, w]$. In the right-hand side of (2.4), $\tau$ applies to the assignments in the *then* and *else* branches of the conditional and, recursively, to nested conditionals. Expressions of the form $\tau(B, [lhs(B)])$ indicate the formal application of the predicate transformer $\tau$ on a *tuple* of variables instead of a proper predicate. The semantics of *let-in* with tuples is such that every variable that is not explicitly assigned a value in the *let* part stays the same: $\$let([x_1, \ldots, x_n] \triangleq [], e)$ is equivalent to $\$let([x_1, \ldots, x_n] \triangleq [x_1, \ldots, x_n], e)$.

---

4Since $\tau$ is applied recursively as usual, consecutive assignments to the same variable translate to nested *let-ins* (see sequential composition and assignments rules).
Finally, let us outline how to transform any conditional into purely active code. Since structured imperative Boogie code can be desugared into assignments (including the nondeterministic assignment havoc), passive statements, and conditionals, we only need to explain how to handle passive statements. The idea is to introduce a fresh Boolean variable \( \alpha \) for every passive statement assume \( b \): set \( \alpha \) to true before the conditional; replace assume \( b \) by \( \alpha := b \); and add assume \( \alpha \) after the conditional. Since \( \alpha \) is fresh, it can be tested after the conditional in any order; since it is initialized to true it does not interfere with the other branch (where the assumption or assertion does not appear). The same approach works for assert passive statements. Overall, this encoding generates VCs of size linear in the size of the input program.

2.4 Implementation and Experiments

**Implementation.** We implemented the translation described in Sec. 2.3 as a command-line tool BLT. BLT is written in Haskell and reuses parts of Boogaloo’s front-end [55] to parse and typecheck Boogie code. The translation is implemented as the composition of a collection of functions, each taking care of the encoding of one Boogie language features; this facilitates extensions and modifications in response to language and translation changes.

BLT inputs a Boogie file, generates its VCs in TPTP, feeds them to Vampire, and reports back the overall outcome. An option is available to choose between the tuple-based (2.4) and the duplication-based (2.3) encoding of conditionals; some experiments, which we describe later, compared the performance of these two encodings.

2.4.1 Experimental subjects

The experiments target Boogie programs in groups demonstrating different traits of the TPTP encoding of VCs and of BLT:

**Group E** consists of examples selected to demonstrate the impact of using triggers, and thus BLT’s capability of handling quantifiers without triggers.

**Group A** is a selection of algorithmic problems (such as searching and sorting), which demonstrates to what extent BLT measures up to Boogie on problems in the latter’s natural domain.

**Group T** is a selection of programs from Boogie’s test suite which demonstrate BLT’s applicability to a variety of features of the Boogie language.

[https://github.com/boogie-org/boogie/tree/master/Test](https://github.com/boogie-org/boogie/tree/master/Test)
**Group S** consists of few Boogie programs with a fixed structure and increasingly larger size, used to assess BLT’s *scalability* and the efficiency of its generated VCs.

We wrote the programs in group $E$ based on examples in the Boogie tutorial and in papers discussing trigger design [44][3][45][56]. We took the programs in group $A$ from our previous work [25], with small changes to fit BLT’s currently supported Boogie features. We retained in group $T$ all test programs that only use language features currently fully supported by BLT, and do not target options or features of the Boogie tool—such as assertion inference or special type encoding—other than vanilla deductive modular verification. We constructed the programs in group $S$ by repeating conditional assignments according to different, repetitive patterns (for example as a sequence of conditional increments to the same variable); the resulting programs allow us to empirically evaluate the size of the VCs generated by BLT, and to what extent Vampire can handle them efficiently. Tab. 2.1 shows some statistics about the size of the programs in each group, as well as that of the VCs generated by Boogie in SMT-LIB, and by BLT in TPTP. BLT’s repository ([https://emptylambda.github.io/BLT/](https://emptylambda.github.io/BLT/)) includes all Boogie programs used in the experiments.

<table>
<thead>
<tr>
<th>GROUP</th>
<th># VCs</th>
<th>BOOGIE (LOC)</th>
<th>SMT-LIB (KBYTES)</th>
<th>TPTP T. (KBYTES)</th>
<th>TPTP D. (KBYTES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>9</td>
<td>19 13 20 49 181</td>
<td>2 3 7 26 1 2 8 15 1 2 8 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td>10</td>
<td>42 17 44 152 439</td>
<td>3 14 80 144 2 17 102 166 2 17 104 172</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>56</td>
<td>279 6 29 137 1614</td>
<td>1 8 93 423 0 3 44 140 2 33 136</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>51</td>
<td>51 6 295 5122 15039</td>
<td>1 31 647 1574 0 68 1832 3493 0 28 10^3 7 10^5 14 10^5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1. Data for the Boogie programs used in the experiments and their translation to TPTP: for each GROUP, how many Boogie programs (#) the group includes, how many verification conditions (VCs) the programs determine in total (in BLT’s encoding); the minimum $m$, mean $\mu$, maximum $M$, and total $\Sigma$ length of the programs in non-comment non-blank lines of code (BOOGIE (LOC)); the minimum $m$, mean $\mu$, maximum $M$, and total $\Sigma$ size in kbytes of the SMT-LIB encoding of the VCs built by Boogie (SMT-LIB), of the TPTP encoding of the VCs built by BLT using tuples (TPTP T.), and using duplication (TPTP D.).

### 2.4.2 Experimental setup

All the experiments ran on a Ubuntu 14.04 LTS GNU/Linux box with Intel 8-core i7-4790 CPU at 3.6 GHz and 16 GB of RAM, with the following tools:

---

6The size of the SMT-LIB encoding gives an idea of the size of the generated VCs, but in the experiments we used Boogie in its default mode where it feeds VCs directly through Z3’s API.

---

26
Boogie 2.3.0.61016, Z3 4.3.2, and Vampire 4.0.

<table>
<thead>
<tr>
<th>GROUP</th>
<th>VCs</th>
<th>BOOGIE (WITH Z3)</th>
<th>BLT T. (WITH VAMPIRE)</th>
<th>BLT D. (WITH VAMPIRE)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓t</td>
<td>✓0 m µ M Σ</td>
<td>✓m µ M Σ</td>
<td>✓d µ Σ</td>
</tr>
<tr>
<td>E</td>
<td>19</td>
<td>16 14 0.7 0.7 6.2</td>
<td>19 0.0 0.1 0.2 0.6</td>
<td>16 0.1</td>
</tr>
<tr>
<td>A</td>
<td>42</td>
<td>42 42 0.7 0.7 6.9</td>
<td>26 0.2 290.5 540.6 2904.7</td>
<td>24 258.1</td>
</tr>
<tr>
<td>T</td>
<td>279</td>
<td>137 137 0.7 0.7 37.6</td>
<td>108 0.0 13.9 301.3 776.0</td>
<td>105 13.3</td>
</tr>
<tr>
<td>S</td>
<td>51</td>
<td>51 51 0.7 1.3 34.8</td>
<td>37 0.0 105.8 300.7 5393.8</td>
<td>48 20.7</td>
</tr>
</tbody>
</table>

Table 2.2. A summary of the experimental comparison between Boogie and BLT: for each GROUP, how many verification conditions (VCs) are to be proved; the number of VCs verified by Boogie with user-defined triggers (✓t) and without triggers or other prover-specific annotations (✓0), and its the minimum m, mean µ, maximum M, and total Σ verification time (without triggers); the number of VCs verified by BLT, and the minimum m, mean µ, maximum M, and its total Σ verification time of the VCs with tuple-based encoding (✓) and with duplication-based encoding (✓d).

Each experiment targets one Boogie program and runs four verification attempts: (i) Boogie runs on b (✓t); (ii) Boogie runs on b with all prover annotations (in particular, triggers) removed (✓0); (iii) BLT runs on b, encoding conditionals using tuples (✓); (iv) BLT runs on b, encoding conditionals using duplication (✓d). We always used Boogie with the /noinfer option, which disables inference of loop invariants; since BLT does not have any inference capabilities, this ensures that we are only comparing their performance of VC generation and checking. We used different timeouts per verification condition in each group—E: 30s; A: 180s; T: 30s; S: 300s—while capping the memory to the available free RAM; BLT may use up to 30s to generate VCs in each problem, although this time is measurable only in group S’S scalability experiments.

Except to specify timeouts and the input format, we always ran Vampire with default options; in particular, we did not experiment with its numerous proof search strategies: while users familiar with Vampire’s internals may be able to tweak them to get better performance in some examples, we want to focus on assessing the predictability of behavior when we use the first-order prover as a black box—in contrast to through lower-level annotations and directives.

2.4.3 Experimental results

Tab. 2.2 shows the number of successful verification attempts in each case, as well as statistics on the wall-clock running time. The most direct comparison

---

7Remember that BLT always ignores triggers and other prover annotations in the Boogie input.
is between $\checkmark_0$ and $\checkmark$, which shows how BLT compares to Boogie without the help of triggers.

The experiments in group $E$ highlight five cases where Boogie’s effectiveness crucially depends on triggers; thus, BLT outperforms Boogie since it can prove all 19 VCs independent of triggers or other quirks of the encoding. The experiments in group $A$ indicate that there remains a considerable effectiveness gap between Boogie and BLT when it comes to algorithmic reasoning, which is mainly due to first-order provers’ still limited capabilities of reasoning about arithmetic and other theories that feature strongly in program correctness; the gap of performance (that is, running time) is instead mainly due to the fact that Vampire continues a proof attempt until reaching the given timeout, whereas Z3 normally terminates quickly. The experiments in group $T$ indicate that BLT provides a reasonably good coverage of the Boogie language, but is sometimes imperfect in reasoning about some features. Note that several of the programs in $T$ are supposed to fail verification, and we observed that BLT’s behavior is consistent on these—that is, it does not produce spurious proofs.\footnote{While the total number of VCs verified by Boogie in group $T$ (137) is the same with ($\checkmark_t$) and without ($\checkmark_0$) prover-specific annotations, the two sets are different: 13 VCs verify without annotations but do not verify with annotations because they correspond to tests that should fail with the annotations; another 13 VCs verify with annotations but not without them.}

**Scalability.** Let us look more closely into the experiments in group $S$, which assess the scalability of BLT, and compare its two encodings—tuple-based (2.4) and duplication-based (2.3)—of conditionals. Boogie scales effortlessly on these examples, so we focus on BLT’s performance.

First, note that the two encodings yield similar performance in the program groups other than $S$, which do not include long sequences of conditional statements. More precisely, group $S$ includes four families of programs; programs in each family have identical structure and different size, determined by a size parameter that grows linearly. Family $S_v$ performs simple assignments on a growing number of variables; family $S_a$ performs a growing number of assignments on a fixed number of variables; families $S_i$ and $S_n$ perform conditional assignments following different patterns—sequential and nested conditionals.

BLT scales as well as Boogie when we increase the number of variables or assignments ($S_v$ and $S_a$): the verification time with both tools is essentially insensitive to input size and under one second per input program. In contrast, BLT’s performance degrades significantly when we increase the number of conditionals, so that group $S$’s numbers in Tab. 2.2 are dominated by the experiments in $S_i$ and $S_n$. Fig. 2.4 illustrates the different behavior of the two encodings in $S_i$ (the results in $S_n$ are qualitatively similar). As expected (Fig. 2.4 left), the tuple-based encoding scales with the input program size,
Figure 2.4. Scalability of BLT on the programs of group $S_i$. Left: how the size of the VCs grows with the input size parameter $n$, in the tuple-based encoding (2.4) and in the duplication-based encoding (2.3). Right: how the verification time of the TPTP VCs grows with the input size parameter $n$, again in each encoding.

whereas the duplication-based encoding blows up exponentially—and in fact the largest example in this group can only be generated with the tuple-based encoding within 30 seconds. The verification time (Fig. 2.4 right) shows a somewhat more unexpected picture: Vampire can digest very large input files, and is generally faster on the wasteful duplication-based encoding; in contrast, reasoning about tuples requires much memory and is quite slow in these conditions. Extrapolating the trends in Fig. 2.4 it seems that the verification time of tuple-based VCs may eventually reach a plateau—even though is currently too large in absolute value to be practical.

We plan to experiment with different encodings of conditional statements to investigate ways of assuaging the current scalability limitations of BLT. It is however encouraging that BLT’s performance on the smaller, yet more logically complex, examples in the other groups is often satisfactory.

### 2.5 Related Work

Triggers were first proposed by Greg Nelson in his influential PhD work [52]. Simplify [17] was the first SMT solver implementing those ideas; today, most widely used SMT solvers—including Z3 [15] and CVC4 [6]—support trigger annotations and include trigger-selection heuristics for when the input does not include such annotations.

As we repeatedly argued in this paper, triggers are indispensable as they increase the flexibility of SMT solvers—especially for program proving—but also introduce an additional annotation burden, and reduce the predictability and
stability of provers. A key challenge in developing program provers based on SMT solvers is designing suitable triggers, but few publications deal explicitly with the problem of trigger selection—which thus remains a skill prohibitively difficult to master. Among these works, Spec# generates special triggers to support list comprehensions in specifications [44]; the Dafny verifier includes flexible strategies to generate triggers that avoid matching loops while also supporting calculations of ground facts from recursive definitions [3]; recently, Dafny has been extended with a mechanism that helps users design triggers in their verified programs [45]. The behavior of triggers has also been analyzed in the context of the VCC [9] and Why3 [19] verifiers.

First-order theorem provers approach the problem of checking validity using techniques, such as saturation, quite different from those of SMT solvers. As a result, they fully support complex usage of quantifiers, but they tend to struggle dealing with theories that are not practical to axiomatize—which has restricted their usage for program verification, where theory reasoning is indispensable for dealing with basic types. The results of the present paper rely on recent developments of the Vampire theorem prover [39], which have significantly extended the support for theory reasoning with a first-class Boolean sort and polymorphic arrays [37].

Others have used the Boogie language as input to tools other than the Boogie verifier, to extend the capabilities of verifiers using Boogie as intermediate representation. HOL-Boogie [8] uses a higher-order interactive prover to discharge Boogie’s verification conditions; Boogaloo [55] and Symbooglix [48] support the symbolic execution of Boogie programs; Boogie2Why [1] translates Boogie into Why3, to take advantage of the latter’s multi-prover support.

### 2.6 Discussion and Future Work

The experimental results detailed in Sec. 3.4 show the feasibility of using a first-order prover for program verification. The gap between BLT and Boogie is still conspicuous—both in applicability and in performance—but we must also bear in mind that most programs used in the experimental evaluation have been written expressly to demonstrate Boogie’s capabilities, and thus it is unsurprising that Boogie works best on them. In Sec. 2.2, however, we have highlighted situations where Boogie’s behavior becomes brittle and dependent on low-level annotations such as triggers; it is in these cases that a different approach, such as the one pursued by BLT, can have an edge—if not yet in overall performance at least in predictability and usability at a higher level.

BLT remains quite limited in scalability and theory reasoning compared to
approaches using SMT solvers. Progress in both areas depends on improvements to the Boogie-to-TPTP encoding, as well as to the back-end prover Vampire. Only recently has Vampire been extended with support [37, 38] for some of the TPTP features that the encoding described in Sec. 2.3 depends on; hence, BLT will immediately benefit from improvements in this area—in particular in the memory-efficiency of rules for tuple reasoning. As future work, we plan to fine-tune the TPTP encoding for performance; the experiments of Sec. 3.4 suggest focusing on finding a scalable encoding of conditionals. There is also room for improving the encoding based on static analysis of the source Boogie code—a technique that is used in different modules of the Boogie tool but not in any way by the current BLT prototype. Finally, we will extend the TPTP encoding to cover the features of the Boogie language currently unsupported—most notably, type polymorphism and gotos.

This paper’s research fits into a broader effort of integrating different verification techniques and tools to complement each other’s shortcoming. Our results suggest that it is feasible to rely on first-order provers to discharge verification conditions in cases where the more commonly used SMT solvers are limited by incompleteness and exhibit brittle behavior, so as to make verification ultimately more flexible and with a higher degree of automation.

Acknowledgments. We thank Evgenii Kotelnikov for helping us understand the latest features of Vampire’s support for FOOL.
Chapter 3

Robustness Testing of Intermediate Verifiers

Yu-Ting Chen, Carlo A. Furia

Abstract. Program verifiers are not exempt from the bugs that affect nearly
every piece of software. In addition, they often exhibit brittle behavior: their
performance verifying a certain program changes considerably with details of
how the input program is expressed—details that should be irrelevant, such as
the order of independent declarations. Such a lack of robustness frustrates users
who have to spend considerable time figuring out a tool’s idiosyncrasies before
they can use it effectively.

This paper introduces a technique to detect lack of robustness of program
verifiers; the technique is lightweight and fully automated, as it is based on testing
methods (such as mutation testing and metamorphic testing). The key idea is
to generate many simple variants of a program that initially passes verification.
All variants are, by construction, equivalent to the original program; thus, any
variant that fails verification indicates lack of robustness in the verifier.

We implemented our technique in a tool called µgie, which operates on
programs written in the popular Boogie language for verification—used as inter-
mediate representation in numerous program verifiers. Program verifiers are
not exempt from the bugs that affect nearly every piece of software. In addition,
they often exhibit brittle behavior: their performance verifying a certain pro-
gram changes considerably with details of how the input program is expressed—
details that should be irrelevant, such as the order of independent declarations.
Such a lack of robustness frustrates users who have to spend considerable time
figuring out a tool’s idiosyncrasies before they can use it effectively.

To appear in proceedings of the 16th International Symposium on Automated
Technology for Verification and Analysis (ATVA 2018),
3.1 Introduction

Automated program verifiers have become complex pieces of software; inevitably, they contain bugs that make them misbehave in certain conditions. *Verification tools need verification too.*

In order to apply verification techniques to program verifiers, we have to settle on the kind of (correctness) properties to be verified. If we simply want to look for basic *programming errors*—such as memory allocation errors, or parsing failures—the usual verification techniques designed for generic software—from random testing to static analysis—will work as well on program verifiers. Alternatively, we may treat a program verifier as a *translator* that encodes the semantics of a program and specification language into purely logic constraints—which can be fed to a generic theorem prover. In this case, we may pursue a correct-by-construction approach that checks that the translation preserves the intended semantics—as it has been done in few milestone research achievements [47, 34].

There is a third kind of analysis, however, which is peculiar to automated program verifiers that aim at being sound. Such tools input a program complete with specification and other auxiliary annotations, and output either “✓ SUCCESS” or “✗ FAILURE”. Success means that the verifier proved that the input program is correct; but failure may mean that the program is incorrect or, more commonly, that the verifier needs more information to verify the program—such as more detailed annotations. This asymmetry between “verified” and “don’t know” is a form of incompleteness, which is inevitable for sound verifiers that target expressive, undecidable program logics. Indeed, using such tools often requires users to become acquainted with the tools’ idiosyncrasies, developing an intuition for what kind of information, and in what form, is required for verification to succeed. To put it in another way, program verifiers may exhibit *brittle, or unstable, behavior*: tiny changes of the input program that ought to be inconsequential have a major impact on the effectiveness achieved by the program verifier. For instance, [Sec. 3.2] details the example of a small program that passes or fails verification just according to the relative order of two unrelated declarations. Brittle behavior of this kind compromises the usability of verification tools.

In this work, we target this kind of *robustness (stability) analysis* of program verifiers. We call an automated verifier *robust* if its behavior is not significantly affected by small changes in the input that should be immaterial. A verifier that is not robust is *brittle* (unstable): it depends on idiosyncratic features of the input. Using brittle verifiers can be extremely frustrating: the feedback we get as

---

1In this paper, the term “verification” also designates *validation* techniques such as testing.
we try to develop a verified program incrementally is inconsistent, and we end up running in circles—trying to fix nonexistent errors or adding unnecessary annotations. Besides being a novel research direction for the verification of verifiers, identifying brittle behavior has the potential of helping develop more robust tools that are ultimately more usable.

More precisely, we apply lightweight verification techniques based on testing. Testing is a widely used technique that cannot establish correctness but is quite effective at finding bugs. The goal of our work is to automatically generate tests that reveal brittleness. Using the approach described in detail in Sec. 3.3 we start from a seed: a program that is correct and can be verified by an automated verifier. We mutate the seed by applying random sequences of predefined mutation operators. Each mutation operator captures a simple variation of the way a program is written that does not change its semantics; for example, it changes the order of independent declarations. Thus, every mutant is a metamorphic transformation of the seed—and equivalent to it. If the verifier fails to verify a mutant we found a bug that exposes brittle behavior: seed and mutant differ only by small syntactic details that should be immaterial, but such tiny details impact the verifier’s effectiveness in checking a correct program.

While our approach to robustness testing is applicable in principle to any automated program verifier, the mutation operators depend to some extent on the semantics of the verifier’s input language, as they have to be semantic preserving. To demonstrate robustness testing in practice, we focus on the Boogie language. Boogie is a so-called intermediate verification language, combining an expressive program logic and a simple procedural programming language, which is commonly used as an intermediate layer in many verification tools. Boogie’s popularity makes our technique (and our implementation) immediately useful to a variety of researchers and practitioners.

As we describe in Sec. 3.3 we implemented robustness testing for Boogie in a tool called $\mu$gie. In experiments described in Sec. 3.4 we ran $\mu$gie on 135 seed Boogie programs, generating and verifying over 87,000 mutants. The mutants triggered brittle behavior in 16 of the seed programs; large, feature-rich programs turned out to be particularly brittle, to the point where several different mutations were capable of making Boogie misbehave. As we reflect in Sec. 3.6 our technique for robustness testing can be a useful complement to traditional testing techniques, and it can help buttress the construction of more robust, and thus ultimately more effective and usable, program verifiers.

**Tool availability.** The tool $\mu$gie, as well as all the artifacts related to its experimental evaluation, are publicly available.
3.2 Motivating Example

Let’s see a concrete example of how verifiers can behave brittlely. Fig. 3.1 shows a simple Boogie program consisting of five declarations, each listed on a separate numbered line.

```boogie
1 function h(int) returns (int);
2 axiom (\forall x, y: int • x > y \implies h(x) > y);
3 const a: [int];
4 axiom (\forall i: int • 0 ≤ i \implies a[i] < a[i+1]);
5 procedure p(i: int) returns (o: int) requires i ≥ 0; ensures o > a[i]; { o := h(a[i+1]); };
```

Figure 3.1. A correct Boogie program that exposes the brittleness of verifiers: changing the order of declarations may make the program fail verification.

The program introduces an integer function h (ln. ??), whose semantics is partially axiomatized (ln. ??); a constant integer map a (ln. ??), whose elements at nonnegative indexes are sorted (ln. ??); and a procedure p (ln. ??, spanning two physical lines in the figure)—complete with signature, specification, and implementation—which returns the result of applying h to an element of a. Never mind about the specific nature of the program; we can see that procedure p is correct with respect to its specification: a[i + 1] > a[i] from the axiom about a and p’s precondition, and thus h(a[i + 1]) > a[i] = o from the axiom about h. Indeed, Boogie successfully checks that p is correct.

There is nothing special about the order of declarations in Fig. 3.1—after all, “the order of the declarations in a [Boogie] program is immaterial” [42, Sec. 1]. A different programmer may, for example, put a’s declarations before h’s. In this case, surprisingly, Boogie fails verification warning the user that p’s postcondition may not hold.\(^2\)

A few more experiments show that there’s a fair chance of running into this kind of brittle behavior. Out of the 5! = 120 possible permutations of the 5 declarations in Fig. 3.1—each an equivalent version of the program—Boogie verifies exactly half, and fails verification of the other half. We could not find any simple pattern in the order of declarations (such as “line x before line y”) that predicts whether a permutation corresponds to a program Boogie can verify.

The SMT-LIB [7] files generated by Boogie (encoding the program’s correctness conditions in a format understood by SMT solvers) also only differ by the order of declarations and assertions—with additional complications due to the fact that declarations have to come before usage in SMT-LIB, and the VCs also include auxiliary functions generated by Boogie. The interplay of Boogie and Z3 determines the brittle behavior we observe in this example: even if it uses a

\(^2\)The first author found out this at the most inappropriate of times—during a live demo!
generic format, Boogie’s SMT-LIB encoding uses plenty of Z3-specific options, but is not always robust in the way it interacts with the solver. To better understand whether other tools’ SMT encodings may be less brittle than Boogie’s, we used \texttt{b2w} \cite{b2w} to translate all 120 permutations of Fig. 3.1 to WhyML—the input language of the Why3 intermediate verifier \cite{why3}. Why3 successfully verified all of them—using Z3 as SMT solver, like Boogie does—which suggests that some features of Boogie’s encoding (as opposed to Z3’s capabilities) are responsible for the brittle behavior on the example.

Such kinds of brittleness—a program switching from verified to unverified based on changes that should be inconsequential—can greatly frustrate users, and in particular novices who are learning the ropes and may get stuck looking for an error in a program that is actually correct—and could be proved so if definitions were arranged in a slightly different way. Since brittleness hinders scalability to projects of realistic size, it can also be a significant problem for advanced users; for example, the developers behind the Ironclad Apps \cite{ironclad} and IronFleet \cite{ironfleet} projects reported\footnote{By an anonymous reviewer of FM 2018.} that “solvers’ instability was a major issue” in their verification efforts.

![Figure 3.2](image_url)

**Figure 3.2.** How robustness testing of Boogie programs works. We start with a correct program $s$ that some Boogie tool $t$ can successfully verify; mutation generator $\mu_{gie}$ mutates $s$ in several different ways, generating many different mutants $m_k$ equivalent to $s$; each mutant undergoes verification with tool $t$; a mutant $m_k$ that \textit{fails verification} with $t$ exposes \textit{brittle behavior} of $t$ on the two equivalent correct programs $s \equiv m_k$.

### 3.3 How Robustness Testing Works

Robustness testing is a technique that “perturbs” a correct and verified program by introducing small changes, and observes whether the changes affect the
program’s verifiability. The changes should be inconsequential, because they are designed not to alter the program’s behavior or specification; if they do change the verifier’s outcome, we found lack of robustness. While robustness testing is applicable to any automated program verifier, we focus the presentation on the popular Boogie intermediate verification language. Henceforth, a “program” is a program (complete with specification and other annotations) written in the Boogie language. Fig. 3.2 illustrates how robustness testing works at a high level; the rest of the section provides details.

In general terms, testing requires to build a valid input, feed it to the system under test, and compare the system’s output with the expected output—given by a testing oracle. Testing the behavior of a verifier according to this paradigm brings challenges that go beyond those involved in generating tests for general programs. First, a verifier’s input is a whole program, complete with specification and other annotations (such as lemmas and auxiliary functions) for verification. Second, robustness testing aims at exposing subtle inconsistencies in a verifier’s output, and not basic programming errors—such as memory access errors, parsing errors, or input/output errors—that every piece of software might be subject to. Therefore, we need to devise suitable strategies for input generation and oracle generation.

### 3.3.1 Mutation Operators

**Input generation.** In order to expose brittleness of verifiers, we need to build complex input programs of significant size, complete with rich specifications and all the annotations that are necessary to perform automated verification. While we may use grammar-based generation techniques [60] to automatically build syntactically correct Boogie programs, the generated programs would either have trivial specifications or not be semantically correct—that is, they would not pass verification. Instead, robustness testing starts from a collection of verified programs—the seeds—and automatically generates simple, semantically equivalent variants of those programs. This way, we can seed robustness testing with a variety of sophisticated verification benchmarks, and assess robustness on realistic programs of considerable complexity.

**Mutation operators.** Given a seed $s$, robustness testing generates many variants $M(s)$ of $s$ by “perturbing” $s$. Building on the basic concepts and terminology of mutation testing [16, 33], we call mutant each variant $m$ of a seed $s$ obtained by applying a random sequence of mutation operators.

---

4 Sec. 3.4.2 describes some experiments with seeds that fail verification. Unsurprisingly, random mutations are unlikely to turn an unverified program into a verified one—therefore, the main paper focuses on using verified programs as seeds.

5 See Sec. 3.5 for a discussion of how robustness testing differs from traditional mutation testing.
A mutation operator captures a simple syntactic transformation of a Boogie program; crucially, mutation operators should not change a program’s semantics but only introduce equivalent or redundant information. Under this fundamental condition, every mutant $m$ of a seed $s$ is equivalent to $s$ in the sense that $s$ and $m$ should both pass (or both fail) verification. This is an instance of metamorphic testing, where we transform between equivalent inputs so that the seed serves as an oracle to check the expected verifier output on all of the seed’s mutants.

Based on our experience using Boogie and working around its brittle behavior, we designed the mutation operators in Table 3.1, which exercise different language features:

<table>
<thead>
<tr>
<th>STRUCTURAL</th>
<th>LOCAL</th>
<th>GENERATIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$ Swap any two declarations</td>
<td>$L_1$ Swap any two local variable declarations</td>
<td>$G_1$ Add $\text{true}$ as pre/postcondition, intermediate assertion, or loop invariant clause</td>
</tr>
<tr>
<td>$S_2$ Split a procedure definition into declaration and implementation</td>
<td>$L_2$ Split a declaration of multiple variables into multiple declarations</td>
<td>$G_2$ Remove a trigger annotation</td>
</tr>
<tr>
<td>$S_3$ Move any declaration into a separate file (and call Boogie on both files)</td>
<td>$L_3$ Join any two preconditions into a conjunctive one</td>
<td></td>
</tr>
<tr>
<td>$L_4$ Join any two postconditions into a conjunctive one</td>
<td>$L_5$ Swap any two pre/postcondition, intermediate assertion, or loop invariant clauses</td>
<td></td>
</tr>
<tr>
<td>$L_6$ Complement an if condition and switch its then and else branches</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1. Mutation operators of Boogie code in categories structural, local, and generative. Operators do not change the semantics of the code they are applied to (except possibly $G_2$ which is used separately).
Generative mutation operators alter redundant information—by adding trivial assertions\(G_1\), and removing quantifier instantiation suggestions (“triggers” in\(G_2\)).

We stress that our mutation operators do not alter the semantics of a Boogie program according to the language’s specification\[42\]: in Boogie, the order of declarations is immaterial\(S_1\),\(L_1\),\(L_2\); a procedure’s implementation may be with its declaration or be separate from it\(S_3\); multiple input files are processed as if they were one\(S_3\); multiple specification elements are implicitly conjoined, and their relative order does not matter\(L_3\),\(L_4\),\(L_5\); a conditional’s branches are mutually exclusive\(L_6\); and true assertions are irrelevant since Boogie only checks partial correctness\(G_1\).

Triggers.\(G_2\) is the only mutation operator that may alter the semantics of a Boogie program in practice: while triggers are suggestions on how to instantiate quantifiers, they are crucial to guide SMT solvers and increase stability in practice\[46, 12\]. Therefore, we do not consider\(G_2\) semantics-preserving: our experiments only apply\(G_2\) in a separate experimental run to give an idea of its impact in isolation.

More mutation operators are possible, but the selection in\Tab. 3.1 should strike a good balance between effectiveness in setting off brittle behavior and feasibility of studying the effect of each individual operator in isolation.

Example. In the example of\Sec. 3.2 the program in\Fig. 3.1 is a possible seed—a correct Boogie program that verifies. Applying mutation operator\(S_1\) twice—first to lines ??, ??, and then to lines ??, ??—generates a mutant where a’s declarations come before h. As discussed in\Sec. 3.2 this mutant fails verification even if it is equivalent to the seed.

3.3.2 Mutation Generation

Given a seed\(s\), the generation of mutants repeatedly draws random mutation operators and applies them to\(s\), or to a previously generated mutant of\(s\), until the desired number\(N_M\) of mutants is reached.

Alg. 1 shows the algorithm to generate mutants. The algorithm maintains a pool\(M\) of mutants, which initially only includes the seed\(s\). Each iteration of the main generation loop proceeds as follows: 1. pick a random program\(p\) in the pool\(M\); 2. select a random mutation operator\(o\); 3. apply\(o\) to\(p\), giving mutant\(m\); 4. add\(m\) to pool\(M\) (if it is not already there).

Users can bias the random selection of mutation operators by assigning a weight\(w(o)\) to each mutation operator\(o\) in\Tab. 3.1: the algorithm draws an operator with probability proportional to its weight, and operators with zero
input : seed program $s$
input : weight $w(o)$ for each mutation operator $o$
input : number of mutants $N_M$
output : set of mutants $M$ of $s$

$M \leftarrow \{s\}$ \hspace{1cm} // initialize pool of mutants to seed
attempts $\leftarrow 0$ \hspace{1cm} // number of main loop iterations

while $|M| < N_M$ do \hspace{1cm} // repeat until $N_M$ mutants are generated

\hspace{1cm} if attempts $> \text{MAX\_ATTEMPTS}$ then
\hspace{2cm} break
end

\hspace{1cm} $p \leftarrow$ any program in $M$
\hspace{1cm} $o \leftarrow$ any mutation operator \hspace{1cm} // draw with probability $w(o)$
\hspace{1cm} $m \leftarrow o(p)$ \hspace{1cm} // apply mutation operator $o$ to $p$
\hspace{1cm} $M \leftarrow M \cup \{m\}$ \hspace{1cm} // add $m$ to pool $M$
\hspace{1cm} attempts $\leftarrow$ attempts + 1

end

return $M$

Algorithm 1: Mutant generation algorithm

weight are never drawn.

Besides the mutation operator selection, there are two other passages of the algorithm where random selection is involved: a program $p$ is drawn uniformly at random from $M$; and applying an operator $o$ selects uniformly at random program locations where $o$ can be applied. For example, if $o$ is $S_1$ (swap two top-level declarations), applying $o$ to $p$ involves randomly selecting two top level declarations in $p$ to be swapped.

Any mutation operator can generate only finitely many mutants; since the generation is random, it is possible that a newly generated mutant is identical to one that is already in the pool. In practice, this is not a problem as long as the seed $s$ is not too small or the enabled operators too restrictive (for example, $S_2$ can only generate $2^D$ mutants, where $D$ is the number of procedure definitions in $s$). The generation loop has an alternative stopping conditions that gives up after MAX\_ATTEMPTS iterations that have failed to generate enough distinct mutants.

Robustness testing. After generating a set $M(s)$ of mutants of a seed $s$, robustness testing runs the Boogie tool on each mutant in $M(s)$. If Boogie can verify $s$ but fails to verify any mutant $m \in M(s)$, we have found an instance of brittle behavior: $s$ and $m$ are equivalent by construction, but the different form in which $m$ is expressed trips up Boogie and makes verification fail on an otherwise correct program.
3.3.3 Implementation

We implemented robustness testing as a commandline tool $\mu$gie (pronounced “moogie”). $\mu$gie implements in Haskell the mutation generation Alg. [1] and extends parts of Boogaloo’s front-end [55] for parsing and typechecking Boogie programs.

Each mutation operator is implemented as a pure function from Boogie programs to Boogie programs. With this design, adding new mutation operators or changing their order of application is straightforward as it just uses Haskell’s function composition. Upon terminating, $\mu$gie outputs each mutant as a separate file (or files, if $S_3$ is applied), and annotates each file with a comment header indicating the seed and the sequence of mutation operators that were applied to generate the mutant from the seed.

**Pretty printing.** $\mu$gie represents programs by their abstract syntax trees, which are then rendered using concrete syntax when all mutants have been generated. Such a pretty printing may introduce small syntactic changes—when the same abstract syntax can be rendered using different syntactic forms. Besides obvious changes in white spaces, line breaks, and comments (which are removed), $\mu$gie’s pretty printer may introduce normalizing changes: functions and procedures without return clause in their signature get an explicit `returns()` clause. We checked that pretty printing has no effect on the behavior of Boogie programs (that is, a pretty-printed seed verifies iff the original seed also verifies).

3.4 Experimental Evaluation

Robustness testing was initially motivated by our anecdotal experience using intermediate verifiers. To rigorously assess to what extent they are indeed brittle, and whether robustness testing can expose their brittleness, we conducted an experimental evaluation using $\mu$gie. This section describes design and results of these experiments.

3.4.1 Experimental Design

A run of $\mu$gie inputs a seed program $s$ and outputs a number of metamorphic mutants of $s$, which are then verified with some tool $t$ (see Fig. 3.2).

**Seed selection.** We prepared a curated collection of seeds by selecting Boogie programs from several different sources, with the goal of having a diverse representation of how Boogie may be used in practice. Each example belongs to

---

$^6$Precisely, the input domain also includes a random seed to select the program elements to which the operator is applied.
one of six groups according to its origin and characteristics; Tab. 3.2a displays basic statistics about them. Group A contains basic Algorithms (search in an array, binary search trees, etc.) implemented directly in Boogie in our previous work [25]; these are relatively simple, but non-trivial, verification benchmarks. Group T is a different selection of mainly algorithmic problems (bubble sort, Dutch flag, etc.) included in Boogie’s distribution Tests. Group E consists of small Examples from our previous work [12] that target the impact of different trigger annotations in Boogie. Group S collects large Boogie programs that we generated automatically from fixed, repetitive structures (for example, nested conditionals); in previous work [12] we used these programs to evaluate Scalability. Groups D and P contain Boogie programs automatically generated by the Dafny [43] and AutoProof [26] verifiers (which use Boogie as intermediate representation). The Dafny and Eiffel programs they translate come from the tools’ galleries of verification benchmarks [14, 4]. As we see from the substantial size of the Boogie programs they generate, Dafny and AutoProof introduce a significant overhead as they include axiomatic definitions of heap memory and complex types. In all, we collected 135 seeds of size ranging from just 6 to over 8 500 lines of Boogie code for a total of nearly 260 000 lines of programs and specifications.

<table>
<thead>
<tr>
<th>GROUP</th>
<th># SEEDS</th>
<th>LOC MIN</th>
<th>MEDIAN</th>
<th>MEAN</th>
<th>MAX</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>17</td>
<td>34</td>
<td>44</td>
<td>152</td>
<td>439</td>
</tr>
<tr>
<td>D</td>
<td>26</td>
<td>2 000</td>
<td>4 076</td>
<td>4 465</td>
<td>8 533</td>
<td>116 101</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
<td>13</td>
<td>18</td>
<td>23</td>
<td>49</td>
<td>230</td>
</tr>
<tr>
<td>P</td>
<td>30</td>
<td>986</td>
<td>1 665</td>
<td>1 911</td>
<td>5 737</td>
<td>57 330</td>
</tr>
<tr>
<td>S</td>
<td>51</td>
<td>6</td>
<td>126</td>
<td>1 047</td>
<td>7 286</td>
<td>67 006</td>
</tr>
<tr>
<td>T</td>
<td>8</td>
<td>11</td>
<td>41</td>
<td>1 662</td>
<td>7 378</td>
<td>18 283</td>
</tr>
<tr>
<td>all</td>
<td>135</td>
<td>6</td>
<td>642</td>
<td>1 718</td>
<td>8 533</td>
<td>259 389</td>
</tr>
</tbody>
</table>

(a) Selection of Boogie programs used as seeds: for each GROUP, the number of programs in that group (# SEEDS), and their MINimum, MEDIAN, MEAN, MAXimum, and TOTAL size in non-blank non-comment lines of code. Row all summarizes measures over all groups.

<table>
<thead>
<tr>
<th>TOOL</th>
<th>COMMIT</th>
<th>DATE</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOGIE 4.1.1</td>
<td>b2d448</td>
<td>2012-09-18</td>
<td>4.1.1</td>
</tr>
<tr>
<td>BOOGIE 4.3.2</td>
<td>97fde1</td>
<td>2015-03-10</td>
<td>4.3.2</td>
</tr>
<tr>
<td>BOOGIE 4.4.1</td>
<td>75b5be</td>
<td>2015-11-19</td>
<td>4.4.1</td>
</tr>
<tr>
<td>BOOGIE 4.5.0</td>
<td>63b360</td>
<td>2017-07-06</td>
<td>4.5.0</td>
</tr>
</tbody>
</table>

(b) Selection of Boogie versions used in the experiments. For every version of the Boogie TOOL, the corresponding COMMIT hash in Boogie’s Git repository, the DATE of the commit, and the matching Z3 version.

Table 3.2. Boogie programs (“seeds”) and Boogie tool versions used in the experiments.

**Tool selection.** In principle, µgie can be used to test the robustness of any verifier that can input Boogie programs: besides Boogie, tools such as Boogaloop [55], Symbooglix [48], and bfft [12]. However, different tools target different kinds of analyses, and thus typically require different kinds of seeds to be tested properly and meaningfully compared. To our knowledge, no tools other than Boogie itself support the full Boogie language, or are as mature and as effective as Boogie for sound verification (as opposed to other analyses, such as
the symbolic execution performed by Boogaloo and Symbooglix) on the kinds of examples we selected. We intend to perform a different evaluation of these tools using µgie in the future, but for consistency and clarity we focus on the Boogie tool in this paper.

In order to understand whether Boogie’s robustness has changed over its development history, our experiments include different versions of Boogie. The Boogie repository is not very consistent in assigning new version numbers, nor does it tag specific commits to this effect. As a proxy for that, we searched through the logs of Boogie’s repository for commit messages that indicate updates to accommodate new features of the Z3 SMT solver—Boogie’s standard and main backend. For each of four major versions of Z3 (4.1.1, 4.3.2, 4.4.1, and 4.5.0), we identified the most recent commit that refers explicitly to that version (see Tab. 3.2b); for example, commit 63b360 says “Calibrated test output to Z3 version 4.5.0”. Then, we call “Boogie v” the version of Boogie at the commit mentioning Z3 version v, running Z3 version v as backend.

To better assess whether brittle behavior is attributable to Boogie’s encoding or to Z3’s behavior, we included two other tools in our experiments: CVC4 refers to the SMT solver CVC4 v. 1.5 inputting Boogie’s SMT2 encoding of verification condition (the same input that is normally fed to Z3); Why3 refers to the intermediate verifier Why3 v. 0.86.3 using Z3 4.3.2 as backend, and inputting WhyML translations of Boogie programs automatically generated by b2w [2].

---

**Table 3.3. Definitions and descriptions of the experimental measures reported in Tab. 3.4.**

<table>
<thead>
<tr>
<th>DEFINITION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>set of all seeds</td>
</tr>
<tr>
<td>$M_O(s)$</td>
<td>set of all mutants of seed $s$ (generated with mutation operators $O$)</td>
</tr>
<tr>
<td>$S_T^*$</td>
<td>seeds that pass verification with tool $t$</td>
</tr>
<tr>
<td>$M_0(s)^\sim\times$</td>
<td>mutants of seed $s$ that fail verification with tool $t$</td>
</tr>
<tr>
<td>$S_T^{\sim\times}$</td>
<td>passing seeds with at least one mutant failing with tool $t$</td>
</tr>
<tr>
<td>$M_0(s)^\sim\times$</td>
<td>failing mutants of seed $s$ that time out with tool $t$</td>
</tr>
<tr>
<td># PASS $</td>
<td>S_T^*</td>
</tr>
<tr>
<td>% FAIL $</td>
<td>S_T^{\sim\times}</td>
</tr>
<tr>
<td>% FAIL $100 \cdot \text{mean}_{s \in S_T^{\sim\times}}</td>
<td>M_O(s)^\times</td>
</tr>
<tr>
<td>% TIMEOUT $100 \cdot \text{mean}_{s \in S_T^{\sim\times}}</td>
<td>M_O(s)^\sim\times</td>
</tr>
<tr>
<td>% FAIL $100 \cdot \text{mean}_{s \in S_T^{\sim\times}}</td>
<td>M_O(s)^\sim\times</td>
</tr>
</tbody>
</table>

**Experimental setup.** Each experiment has two phases: first, generate mutants for every seed; then, run Boogie on the mutants and check which mutants still verify.

For every seed $s \in S$ (where $S$ includes all 135 programs summarized in
Tab. 3.2a), we generate different batches $M_{O}(s)$ of mutants of $s$ by enabling specific mutation operators $O$ in $\mu$gie. Precisely, we generate 12 different batches for every seed:

$M_*(s)$ consists of 100 different mutants of $s$, generated by picking uniformly at random among all mutation operators in Tab. 3.1 except $G_2$ (that is, each mutation operator gets the same positive weight, and $G_2$ gets weight zero);

$M_J(s)$, for $J$ one of the 11 operators in Tab. 3.1, consists of 50 different mutants of $s$ generated by only applying mutation operator $J$ (that is, $J$ gets a positive weight, and all other operators get weight zero).

Batch $M_*$ demonstrates the effectiveness of robustness testing with general settings; then, the smaller batches $M_J$ focus on the individual effectiveness of one mutation operator at a time. Operator $G_2$ is only used in isolation (and not at all in $M_*$) since it may change the semantics of programs indirectly by guiding quantifier instantiation.

Let $t$ be a tool (a Boogie version in Tab. 3.2b, or another verifier). For every seed $s \in S$, we run $t$ on $s$ and on all mutants $M_{O}(s)$ in each batch. For a run of $t$ on program $p$ (seed or mutant), we write $t(p)$ if $t$ verifies $p$ successfully; and $\neg t(p)$ if $t$ fails to verify $p$ (because it times out, or returns with failure). Based on this basic data, we measure robustness by counting the number of verified seeds whose mutants fail verification: see the measures defined in Tab. 3.3 and the results described in detail in Sec. 3.4.2. As an additional check that the mutation operators do not change the semantics of programs, we ascertained that all mutants pass parsing and other syntactic checks.

To reduce the running time in practical usage scenarios, one may stop verification of a seed’s mutants as soon as one of them fails verification, or even alternate seed mutation and verification steps to avoid generating mutants after the first failure. In this paper’s experiments, however, we decided to run verification exhaustively on every mutant in order to collect detailed information about the effectiveness of robustness testing.

**Running times.** The experiments ran on a Ubuntu 16.04 LTS GNU/Linux box with Intel 8-core i7-4790 CPU at 3.6 GHz and 16 GB of RAM. Generating the mutants took about 15 minutes for the batch $M_*$ and 10 minutes for each batch $M_J$. Each verification run was given a timeout of 20 seconds, after which it was forcefully terminated by the scheduler of GNU parallel [59]. For replicability, seeds and summary output of the experimental runs are available online [51].
Table 3.4. Experimental results of robustness testing with \( \mu \). For each group of seeds, for each tool: number of seeds passing verification (# PASS), number and percentage of passing seeds for which at least one mutation fails verification (# \( \exists \) FAIL and % \( \exists \) FAIL), average percentage of mutants per passing seed that fail verification (% \( \exists \) FAIL), average percentage of mutants per passing seed that time out (% \( \exists \) TIMEOUT), average percentage of mutants that fail verification per passing seed with at least one failing mutant (% \( \exists \) FAIL). The middle section of the table records experiments with batch \( M_s \); each of the 11 rightmost columns records experiments with batch \( M_f \), for \( J \) one of the mutation operators in Tab. 3.1.

### 3.4.2 Experimental Results

**Overall results:** batch \( M_s \). Our experiments, whose detailed results are in Tab. 3.4, show that robustness testing is effective in exposing brittle behavior,
which is recurrent in Boogie: for 12% of the seeds that pass verification, there is at least one mutant in batch $M_*$ that fails verification.

Not all seeds are equally prone to brittleness: while on average only 3% of one seed’s mutants fail verification, it is considerably easier to trip up seeds that are susceptible to brittle behavior (that is such that at least one mutant fails verification): 27% of mutants per such seeds fail verification.

When the verifier times out on a mutant, it may be because: \(i\) the timeout is itself unstable and due to random noise in the runtime environment; \(ii\) the mutant takes longer to verify than the seed, but may still be verified given longer time; \(iii\) verification time diverges. We ruled out \(i\) by repeating experiments 10 times, and reporting a timeout only if all 10 repetitions time out. Thus, we can generally consider the timeouts in Tab. 3.4 indicative of a genuine degrading of performance in verification—which affected 3% of one seed’s mutants on average.

**Boogie versions.** There is little difference between Boogie versions, with the exception of Boogie 4.1.1. This older version does not support some language features used extensively in many larger examples that also tend to be more brittle (groups D and P). As a result, the percentage of verified seeds with mutants that fail verification is spuriously lower (4%) but only because the experiments with Boogie 4.1.1 dodged the harder problems and performed similarly to the other Boogie versions on the simpler ones. Leaving the older Boogie version 4.1.1 aside, our experiments leave open the question of whether robustness has significantly improved in recent versions of Boogie.

**Intermediate verifier vs. backend.** Is the brittleness we observed in our experiments imputable to Boogie or really to Z3? To shed light on this question, we tried to verify every seed and mutant using CVC4 instead of Z3 with Boogie’s encoding; and using Why3 on a translation \[2\] of Boogie’s input. Since the seeds are programs optimized for Boogie verification, CVC4 and Why3 can correctly process only about half of the seeds that Boogie can. This gives us too little evidence to answer the question conclusively: while both CVC4 and Why3 seem to be more robust than Boogie, they can verify none of the brittle seeds (that is, verified seeds with at least one failing mutant), and thus behave as robustly as Boogie on the programs that both tools can process\[8\] As suggested by the simple example of Sec. 3.2 (where Why3 was indeed more robust than Boogie), it is really the interplay of Boogie and Z3 that determines brittle behavior. While SMT solvers have their own quirks, Boogie is meant to provide a stable intermediate layer; in all, it seems fair to say that Boogie is at least partly

\[\text{Footnote 7: For clarity, we initially focus on Boogie 4.5.0, and later discuss differences with other versions.}\]

\[\text{Footnote 8: Additionally, Why3 times out on 51 mutants of 2 seeds in group S; this seems to reflect an ineffective translation performed by b2w \[2\] rather than brittleness of Why3.}\]
responsible for the brittleness.

**Program groups.** Robustness varies greatly across groups, according to features and complexity of the seeds that are mutated. Groups D and P are the most brittle: about 1/3 of passing seeds in D, and about 2/5 of passing seeds in P, have at least one mutant that fails verification. Seeds in D and P are large and complex programs generated by Dafny and AutoProof; they include extensive definitions with plenty of generic types, complex axioms, and instantiations. The brittleness of these programs reflects the hardness of verifying strong specifications and feature-rich programming languages: the Boogie encoding must be optimized in every aspect if it has to be automatically verifiable; even a modicum of clutter—introduced by \( \mu \)lities—may jeopardize successful verification.

By the same token, groups A, E, and T’s programs are more robust because they have a smaller impact surface in terms of features and size. Group S’s programs are uniformly robust because they have simple, repetitive structure and weak specifications despite their significant size.

**Mutation operators and batches** \( M_J \). [Fig. 3.3] and the rightmost columns of [Tab. 3.4] explore the relative effectiveness of each mutation operator. \( S_2 \), \( L_1 \), \( L_2 \) and \( L_6 \) could not generate any failing mutant—suggesting that Boogie’s encoding of procedure declarations, of local variables, and of conditionals is fairly robust. In contrast, all other operators could generate at least one failing mutant; [Fig. 3.3] indicates that \( L_3 \) and \( S_3 \) generated failing mutants for respectively 2 seeds and 1 seed that were robust in batch \( M \) (using all mutation operators with the same frequency)—indicating that mutation operators are complementary to a certain extent in the kind of brittleness they can expose. Since every seed that \( M \) trips up can also be tripped up by a single operator, combining multiple mutation operators does not seem to be necessary for successful robustness testing (although predicting which operators will be effective may be hard).

[Tab. 3.5] gives information about the frequency of different mutation operators in failing mutants of batch \( M \). Differences are consistent with the results of batches \( M_J \)—which provide more direct evidence. As usual, the numbers for Boogie 4.1.1 are different because of a much smaller number of passing seeds on which the statistics are computed, which in turn is a result of Boogie 4.1.1’s more limited supported features.

**Failures.** Overall, 13 brittle seeds are revealed by 350 failing mutants in \( M \) with Boogie 4.5.0. Failures are of three kinds: a) timeouts (6 seeds, 252 mutants); b) type errors (5 seeds, 10 mutants); c) explicit verification failures (2 seeds, 88 mutants).

**Timeouts** mainly occur in group D (5 seeds), where size and complexity of the code are such that any mutation that slows down verification may hit the
Figure 3.3. For each of 16 verified seeds with at least one failing mutant with Boogie 4.5.0, which batches all exclusively include a failing mutant of those seeds, $G_2$ is excluded and analyzed separately; $S_2$, $L_2$, $L_6$ could not generate any failing mutant; $L_4$ generated failing mutants for a strict subset of those in $M^*$; $G_1$ generated failing mutants for a strict subset of those in $M^*$.

Timeout limit; verification of some mutants seems to be non-terminating, whereas others are just slowed down by some tens of seconds. One exception is BQueue in group T, whose implementation of a queue in the style of dynamic frames is not particularly large (322 lines) but includes many assertions that take time to verify. Some mutants verify if given longer time; in fact, group T’s programs are otherwise very robust, probably because they are part of Boogie’s test suite, and thus any change in Boogie is checked against the same examples to ensure they still verify.

Type errors all occur in group P and only when mutation $S_3$ splits the seed in a way that procedure update_heap (part of AutoProof’s heap axiomatization) ends up being declared after its first usage; in this case, Boogie cannot correctly instantiate the procedure’s generic type, which triggers a type error even before Z3 is involved. Even though AutoProof’s heap encoding is based on Dafny’s and hence somewhat similar to it, Dafny is immune to such faulty behavior.

Verification failures occur in seeds of group A and D. In particular, a binary search tree implementation in group A fails verification when the relative order of two postconditions is swapped by $L_5$ while Why3 cannot prove the whole example, it can prove the brittle procedure alone regardless of the postcondition order. Group D’s solution to problem 3 in the VerifyThis 2015 competition $[32]$ fails verification when two preconditions are merged into a conjunctive one by $L_3$. 

50
As we expected from previous work [46], altering triggers is likely to make analysis of mutation operators in batch verification fail (30 seeds and 276 mutants overall; 20 seeds are only brittle if triggers are modified); most of these failures (26 seeds and 250 mutants) are brittle if triggers are modified.

Remember that mutation operator $G_2$ is the only one that modifies triggers, and was only applied in isolation in a separate set of experiments. As we expected from previous work [46], altering triggers is likely to make verification fail (30 seeds and 276 mutants overall; 20 seeds are only brittle if triggers are modified); most of these failures (26 seeds and 250 mutants) are brittle if triggers are modified.

In all, it is clear that Boogie’s encoding is quite sensitive to the order of declarations and assertions even when it should not matter.

**Triggers.** Remember that mutation operator $G_2$ is the only one that modifies triggers, and was only applied in isolation in a separate set of experiments. As we expected from previous work [46], altering triggers is likely to make verification fail (30 seeds and 276 mutants overall; 20 seeds are only brittle if triggers are modified); most of these failures (26 seeds and 250 mutants) are brittle if triggers are modified.

---

**Table 3.5.** Analysis of mutation operators in batch $M_\pi$. For each GROUP of seeds, for each TOOL, for each mutation operator OP, column OP reports the percentage of failing mutants of passing seeds that were generated by applying one or more times operator OP (possibly in combination with other mutation operators).
Timeouts, since removing triggers is likely to at least slow down verification—if not make it diverge. Operator \(G_2\) is very effective at exposing brittleness mainly with the complex examples in groups D and P, which include numerous axioms and extensive quantification patterns. Group E’s programs are a bit special because they are brittle—they are designed to be so—but are only affected by mutation operators that remove the trigger annotations on which they strongly depend; in contrast, they are robust against all other mutation operators.

**Failing seeds.** Tab. 3.6 is the counterpart of Tab. 3.4 showing if random mutations may change a seed that fails verification into one that passes it. Unsurprisingly, this does not happen very often: there are only 2 seeds that go from failing to passing with random mutations. One in group E is similar to the example of Sec. 3.2 but where the seed’s order of declarations fails verification, and swapping two of them restores verifiability; one in group P is `sum_and_max.bpl`, which robustly verifies with Boogie 4.3.2 but times out with more recent Boogie versions. AutoProof’s encoding was fine-tuned based on Boogie 4.3.2, which explains why it may be sensitive to using newer Boogie versions.

### 3.5 Related Work

**Robustness.** This paper’s robustness testing aims at detecting so-called *butterfly effects* [46]—macroscopic changes in a verifier’s output in response to minor modifications of its input. Program provers often incur volatile behavior because they use automated theorem provers—such as SMT solvers—which in turn rely on heuristics to handle efficiently, in many practical cases, complex proofs in undecidable logics. Matching triggers—heuristics to guide quantifier instantiation—are especially prone to misfire in response to tiny changes in the input, as observed in previous work [46, 12] and confirmed by our experiments in Sec. 3.4.2.

The notion of robustness originates from dynamical systems theory [24, 36]. While robustness is well understood for linear systems, *nonlinear* systems may manifest unpredictable loss of robustness that are hard to analyze and prevent. In this context, real time temporal logics have been proposed as a way of formalizing and analyzing behavioral properties that are satisfied robustly [20].

**Random testing.** Our approach uses *testing* to expose brittle behavior of verifiers. While testing can only try out finitely many inputs—and thus can only prove the presence of errors, as remarked in one of Dijkstra’s most memorable quotes [18]—it is an invaluable analysis techniques, which requires relatively little effort to be applied. By automatically generating test inputs, *random testing* has proved to be extremely effective at detecting subtle errors in pro-
Table 3.6. For each group of seeds, for each tool: number of seeds failing verification (# FAIL), number and percentage of failing seeds for which at least one mutant passes verification (#/% ∃ PASS), average percentage of mutants per failing seed that pass verification (% PASS), average percentage of mutants that pass verification per failing seed with at least one passing mutant (% PASS). All data in the table is about experiments with batch $M_*$.

<table>
<thead>
<tr>
<th>GROUP</th>
<th>TOOL</th>
<th># FAIL</th>
<th># %PASS</th>
<th>% PASS</th>
<th>% %PASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4.1.1</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>4.3.2</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>4.4.1</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>4.5.0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>CVC4</td>
<td>4</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>WHY3</td>
<td>3</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>D</td>
<td>4.1.1</td>
<td>26</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>4.3.2</td>
<td>2</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>4.4.1</td>
<td>2</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>4.5.0</td>
<td>2</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>CVC4</td>
<td>26</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>WHY3</td>
<td>26</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>E</td>
<td>4.1.1</td>
<td>5</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>4.3.2</td>
<td>3</td>
<td>1</td>
<td>33%</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>4.4.1</td>
<td>3</td>
<td>1</td>
<td>33%</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>4.5.0</td>
<td>3</td>
<td>1</td>
<td>33%</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>CVC4</td>
<td>6</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>WHY3</td>
<td>3</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>P</td>
<td>4.1.1</td>
<td>30</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>4.3.2</td>
<td>16</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>4.4.1</td>
<td>17</td>
<td>1</td>
<td>6%</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>4.5.0</td>
<td>17</td>
<td>1</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>CVC4</td>
<td>30</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>WHY3</td>
<td>30</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>S</td>
<td>4.1.1</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>4.3.2</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>4.4.1</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>4.5.0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>CVC4</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>WHY3</td>
<td>11</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>T</td>
<td>4.1.1</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>4.3.2</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>4.4.1</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>4.5.0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>CVC4</td>
<td>4</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>WHY3</td>
<td>5</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>all</td>
<td>4.1.1</td>
<td>61</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>4.3.2</td>
<td>21</td>
<td>1</td>
<td>5%</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>4.4.1</td>
<td>22</td>
<td>2</td>
<td>9%</td>
<td>2%</td>
</tr>
<tr>
<td></td>
<td>4.5.0</td>
<td>22</td>
<td>2</td>
<td>9%</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>CVC4</td>
<td>70</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>WHY3</td>
<td>78</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Grams completely automatically. Random testing can generate instances of complex data types by recursively building them according to their inductive structure—as it has been done for functional [13, 21] and object-oriented [53, 50] programming languages. Random testing has also been successfully applied to security testing—where it is normally called “fuzzing” [27]—as well as to compiler testing [60, 40]—where well-formed programs are randomly generated.
Mutation testing. This paper’s robustness testing is a form of random testing, in that it applies random mutation operators to transform a program into an equivalent one. The terminology and the idea of applying mutation operators to transform between variants of a program come from mutation testing [33]. However, the goals of traditional mutation testing and of this paper’s robustness testing are specular. Mutation testing is normally used to assess the robustness of a test suite—by applying error-inducing mutations to correct programs, and ascertaining whether the tests fail on the mutated programs. In contrast, we use mutation testing to assess the robustness of a verifier—by applying semantic-preserving mutations to correct (verified) programs, and ascertaining whether the mutated programs still verify. Therefore, the mutation operators of standard mutation testing introduce bugs in a way that is representative of common programming mistakes; the mutation operators of robustness testing (see Tab. 3.1) do not alter correctness but merely represent alternative syntax expressing the same behavior in a way that is representative of different styles of programming.

Metamorphic testing. In testing, generating inputs is only half of the work; one also has to compare the system’s output with the expected output to determine whether a test is passing or failing. The definition of correct expected output is given by an oracle [5]. The more complex the properties we are testing for, the more complex the oracle: a crash oracle (did the program crash?) is sufficient to test for simple errors such as out-of-bound memory access; finding more complex errors requires some form of specification [31] of expected behavior—for example in the form of assertions [10, 54].

Even when directly building an oracle is as complex as writing a correct program, there are still indirect ways of extrapolating whether an output is correct. In differential testing [49], there are variants of the program under test; under the assumption that not all variants have the same bugs, one can feed the same input to every variant, and stipulate that the output returned by the majority is the expected one—and any outlier is likely buggy. Differential testing has been applied to testing compilers [60], looking for the compiler that generates the executable that behaves differently from the others on the same input. In metamorphic testing [11, 57], an input is transformed into an equivalent one according to metamorphic relations (for example, the inputs $x$ and $-x$ should be equivalent inputs to a function computing the absolute value); equivalent inputs that determine different outputs are indicative of error. Our robustness testing applies mutation operators that determine identity metamorphic relations between Boogie programs, since they only change syntactic details and not the semantics of programs.
3.6 Discussion and Future Work

Our experiments with µgie confirm the intuition—bred by frequently using it in our work—that Boogie is prone to brittle behavior. How can we shield users from this brittle behavior, thus improving the usability of verification technology?

Program verifiers that use Boogie as an intermediate representation achieve this goal to some extent: the researchers who built the verifiers have developed an intuitive understanding of Boogie’s idiosyncrasies, and have encoded this informal knowledge into their tools. End users do not have to worry about Boogie’s brittleness but can count on the tools to provide an encoding of their input programs that has a good chance of being effective. In contrast, developers of program verifiers still have to know how to interact with Boogie and be aware of its peculiarities.

Robustness testing may play a role not only in exposing brittle behavior—the focus of this paper—but in precisely tracking down the sources of brittleness, thus helping to debug them. To this end, we plan to address minimization and equivalency detection of mutants in future work. The idea is that the number of failing mutants that we get by running µgie are not directly effective as debugging aids, because it takes a good deal of manual analysis to pinpoint the precise sources of failure in large programs with several mutations. Instead, we will apply techniques such as delta debugging [61] to reduce the size of a failing mutant as much as possible while still triggering failing behavior in Boogie. Failing mutants of minimal size will be easier to inspect by hand, and thus will point to concrete aspects of the Boogie translation that could be made more robust.

To further investigate to what extent it is Z3 that is brittle, and to what extent it is Boogie’s encoding of verification condition—an aspect only partially addressed by this paper’s experiments—we will apply robustness testing directly to SMT problems, also to understand how Boogie’s encoding can be made more robust.

Robustness testing could become a useful help to developers of program and intermediate verifiers, to help them track down sources of brittleness during development, ultimately making verification technology easier to use and more broadly applicable.
Bibliography

— One citation on page 30

— 3 citations on pages 38, 45, and 48

— 3 citations on pages 18, 26, and 30

— One citation on page 44

— One citation on page 54

— One citation on page 29

— One citation on page 37

[8] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelli-


Dafny examples and tests. https://github.com/Microsoft/dafny/tree/master/Test.


Dafny examples and tests. https://github.com/Microsoft/dafny/tree/master/Test.

— One citation on page 4

— One citation on page 10

— One citation on page 30

— One citation on page 30

— One citation on page 30

— One citation on page 54

— 2 citations on pages 36 and 54

— 3 citations on pages 41, 44, and 52

— One citation on page 53

— One citation on page 44


— One citation on page 24

— One citation on page 52

— 2 citations on pages 26 and 44

— One citation on page 44

— One citation on page 4

— One citation on page 53

— One citation on page 22

— One citation on page 7

[34] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet:
— One citation on page 38

— One citation on page 38

— One citation on page 54

— One citation on page 5

— One citation on page 50

— 2 citations on pages 39 and 54

— One citation on page 35

— One citation on page 16

— One citation on page 52
— 5 citations on pages 16, 19, 21, 30, and 31

— One citation on page 31

— One citation on page 30

— One citation on page 53

— 2 citations on pages 23 and 24

[48] K. Rustan M. Leino. This is Boogie 2, 2008. [http://goo.gl/0sH6g](http://goo.gl/0sH6g).  
— 6 citations on pages 3, 6, 8, 36, 37, and 41

— 2 citations on pages 6 and 44

— 3 citations on pages 18, 26, and 30

— 2 citations on pages 4 and 10

— 4 citations on pages 3, 41, 51, and 52
   — 3 citations on pages 18, 26, and 30

   — One citation on page 35

   — 2 citations on pages 30 and 44

   — One citation on page 54

   — One citation on page 53

[58] µgie repository. https://emptylambda.github.io/mu-gie/
   — 2 citations on pages 36 and 46

   — 2 citations on pages 3 and 5

   — One citation on page 29

   — One citation on page 53

   — One citation on page 54
   — 4 citations on pages 25, 30, 43, and 44

   — One citation on page 26

   — One citation on page 54

   — One citation on page 16

   — One citation on page 46

   — One citation on page 6

   — 3 citations on pages 39, 53, and 54

   — One citation on page 55