Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language

GAURAV PARTHASARATHY, ETH Zurich, Switzerland
THIBAULT DARDINIER, ETH Zurich, Switzerland
BENJAMIN BONNEAU, Université Grenoble Alpes - CNRS - Grenoble INP - VERIMAG, France
PETER MÜLLER, ETH Zurich, Switzerland
ALEXANDER J. SUMMERS, University of British Columbia, Canada

Automated program verifiers are typically implemented using an intermediate verification language (IVL), such as Boogie or Why3. A verifier front-end translates the input program and specification into an IVL program, while the back-end generates proof obligations for the IVL program and employs an SMT solver to discharge them. Soundness of such verifiers therefore requires that the front-end translation faithfully captures the semantics of the input program and specification in the IVL program, and that the back-end reports success only if the IVL program is actually correct. For a verification tool to be trustworthy, these soundness conditions must be satisfied by its *actual implementation*, not just the program logic it uses.

In this paper, we present a novel validation methodology that, given a formal semantics for the input language and IVL, provides formal soundness guarantees for front-end implementations. For each run of the verifier, we automatically generate a proof in Isabelle showing that the correctness of the produced IVL program implies the correctness of the input program. This proof can be checked independently from the verifier, in Isabelle, and can be combined with existing work on validating back-ends to obtain an end-to-end soundness result. Our methodology based on forward simulation employs several modularisation strategies to handle the large semantic gap between the input language and the IVL, as well as the intricacies of practical, optimised translations. We present our methodology for the widely-used Viper and Boogie languages. Our evaluation shows that it is effective in validating the translations performed by the existing Viper implementation.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; *Semantics*; • **Theory of computation** → **Program verification**.

Additional Key Words and Phrases: Software Verification, Intermediate Verification Languages, Formal Semantics, Proof Certification

---

Authors' addresses: Gaurav Parthasarathy, ETH Zurich, Department of Computer Science, Zurich, Switzerland, gaurav.parthasarathy@inf.ethz.ch; Thibault Dardinier, ETH Zurich, Department of Computer Science, Zurich, Switzerland, thibault.dardinier@inf.ethz.ch; Benjamin Bonneau, Université Grenoble Alpes - CNRS - Grenoble INP - VERIMAG, Grenoble, France, Benjamin.Bonneau@univ-grenoble-alpes.fr; Peter Müller, ETH Zurich, Department of Computer Science, Zurich, Switzerland, peter.mueller@inf.ethz.ch; Alexander J. Summers, University of British Columbia, Vancouver, Canada, alex.summers@ubc.ca.

---

1 INTRODUCTION

Program verifiers are tools that try to automatically establish the correctness of an input program with respect to a specification. A standard approach for achieving automation is to reduce the input program and specification to a set of first-order formulas whose validity implies the correctness of the input program; the validity of formulas is automatically checked using an SMT solver. Instead of directly producing logical formulas, many program verifiers are *translational verifiers*: they translate an input program and specification into a program in an *intermediate verification language (IVL)*; we call this a *front-end translation*. An IVL comes with its own *back-end verifier* that ultimately reduces IVL programs to logical formulas. This translational approach via an IVL allows for the reuse of the IVL's back-end technology across multiple front-end verifiers, and makes for a more understandable target representation than direct mappings to logical formulas, simplifying the development of state-of-the-art program verifiers.

A very wide variety of practical program verifiers are translational verifiers; e.g. Corral [26], Dafny [29], SMACK [7], SYMDIFF [25], and Viper [34] target the imperative Boogie IVL [28], while Creusot [10] and Frama-C [23] translate to the functional Why3 IVL [17]. Multiple layers of front-end translations and IVLs can also be *composed* (e.g. Prusti [2] builds on Viper as an IVL).

To ensure that successful verification indeed implies that the input program satisfies its specification, any translational verifier must meet two *soundness conditions*: (1) *Front-end soundness*: the *translation* into the IVL is faithful, i.e. correctness of the produced IVL program implies correctness of the input program, and (2) *IVL back-end soundness*: if the back-end IVL verifier reports success, the IVL program is correct. Trustworthiness of program verifiers requires formal guarantees for both soundness conditions. It is *not* sufficient to prove soundness of the program logics they employ in principle: automated verifiers are complex systems, and it is essential that formal guarantees also cover their *actual implementations*, where soundness bugs can and do arise.

Existing work on ensuring front-end soundness is based on idealised implementations that are formalised on paper or in an interactive theorem prover. In practice, practical front-end translations are implemented in efficient mainstream programming languages, use diverse libraries and programming paradigms, and include subtle optimisations omitted from idealised implementations; there is a very large gap between the translations proved correct and the actual translations used in practice. In this paper, we bridge this gap for the first time, developing an approach to formally validate the front-end soundness of translations used in *existing, practical* verifier implementations. IVL back-end verifier soundness, which includes the soundness of the underlying SMT solver, is a better-studied and orthogonal concern; our results can be combined with work in that area to obtain end-to-end guarantees for an entire verification toolchain [5, 16, 18, 19, 39].

Proving front-end soundness once and for all for a realistic verifier implementation is practically infeasible, since such implementations are large (e.g. 17.2 KLOC and 8.5 KLOC for the Dafny-to-Boogie and Viper-to-Boogie front-ends, respectively) and are typically written in languages that lack a full formalisation (C# and Scala, in the examples above). Instead, we develop a translation validation approach that, given a formal semantics for the input language and IVL, *automatically* generates a formal proof on every run of the verifier via an instrumentation of the existing implementation. Our proofs are expressed in the Isabelle theorem prover [35], and thus can be checked independently, effectively removing the (substantial) front-end translation from the trusted code base of the verifier.

Challenges. Formally validating front-end translations is challenging for three main reasons:

1. Semantic gap: There is a large semantic gap between a front-end language and an IVL, which concerns the state model (e.g. neither Boogie nor Why3 have a heap, but most front-end languages do), the execution model (e.g. Viper heap accesses are partial operations that must be guarded by semantic conditions ultimately checked by verification, while Boogie and Why3 use syntactic checks

to guard state accesses such as disallowing global variables in Boogie axioms and restricting aliasing between mutable variables in Why3), and the program logics used to reason about programs (e.g. front-ends use complex logics, such as dynamic frames [22] in Dafny, a flavour of separation logic [36, 41] in Viper, and prophetic reasoning in Creusot [10], whereas Boogie and Why3 do not have built-in support for such logics). To bridge the semantic gap, front-ends translate input programs into a complex combination of low-level operations and background logical axiomatisations of input language concepts; validation needs to precisely account for the combination of these ingredients, while allowing the separation of translation aspects for the sake of modularity and maintainability.

2. Diverse translations: Practical front-end translations are *diverse* in the sense that they use multiple alternative translations for the same feature, e.g. more efficient translations that are sound only in certain cases. These translations also evolve frequently over time, as new techniques and features are developed or optimised; ideally a formal approach to validation should provide means of minimising the impact of the exchange of one translation for another.

3. Non-locality: The soundness of the translation of a fragment of the input program may depend on several checks that are performed at different places in the IVL program. For instance, the translation of a procedure call might be sound only because well-formedness of the procedure specification has been checked elsewhere in the generated IVL code. Such non-local checks are commonly used to speed up verification, for instance, to check well-formedness conditions once and for all rather than each time a specification is used. However, they complicate the soundness argument, which needs to somehow track the dependencies on properties checked elsewhere.

This paper. We present the first approach for enabling automatic formal validation for existing implementations of the front-end translations employed in many practical program verifiers. This validation guarantees front-end soundness and, thus, makes automated program verifiers substantially more trustworthy.

The core of our approach is a general methodology for generating *forward simulations* [32] between the statements of the input and the IVL program in a modular way. Our methodology provides solutions to the three challenges above. It (1) bridges the semantic gap with a novel approach by which the simulation proof is split into smaller simulations, (2) supports diverse translations by expressing simulations abstractly, and (3) handles non-locality by systematically and formally tracking dependencies during a simulation proof.

For concreteness, we present our methodology for the translation from a core fragment of Viper to Boogie, as implemented in an existing and actively-used verification tool [12]. This translation is significant because it exhibits all of the challenges discussed above and because both Viper and Boogie are widely used. For instance, Viper is used in Gobra (Go) [47], Prusti (Rust) [2], Nagini (Python) [14], VerCors (Java) [4], and Gradual C0 [13]. The soundness of each of these tools relies on the Viper verifiers being sound. Note that these tools use Viper as an IVL, but for the purpose of this paper, we will treat it as a front-end language that is translated to Boogie. While our methodology is phrased in terms of Viper and Boogie, we have designed our approach, which solves the key challenges above, to generalise to other front-end translations (e.g. the Dafny-to-Boogie translation).

Contributions. We make the following technical contributions:

- We develop a general methodology for the automatic validation of front-end translations based on forward simulation proofs. We present this methodology for the translation from Viper to Boogie. As a foundation for the proofs, we formalise a semantics for a core subset of Viper in Isabelle and connect this with an existing Isabelle formalisation for Boogie [39].
- We instrument the existing Viper-to-Boogie implementation such that, for a subset of Viper, it automatically generates an Isabelle proof justifying the soundness of the translation.

$VExpr \ni e ::= x \mid lit \mid e.f \mid e\ bop\ e \mid uop(e)$ $VAssert \ni A ::= e \mid \textbf{acc}(e.f, e) \mid A * A \mid e \Rightarrow A \mid e\ ?\ A : A$

$VStmt \ni s ::= x := e \mid e.f := v \mid \vec{y} := m(\vec{x}) \mid m(\vec{x}) \mid \textbf{var}\ x : \tau \mid \textbf{inhale}\ A \mid \textbf{exhale}\ A \mid \textbf{assert}\ A \mid$
$\qquad\qquad s; s \mid \textbf{if}(e)\ \{s\}\ \textbf{else}\ \{s\}$

$BExpr \ni e_b ::= x \mid lit_b \mid e_b\ bop\ e_b \mid uop(e_b) \mid f[\vec{\tau_b}](\vec{e_b}) \mid \forall x : \tau_b.\ e_b \mid \exists x : \tau_b.\ e_b \mid \forall_{ty}\ t.\ e_b \mid \exists_{ty}\ t.\ e_b$

$BSimpleCmd \ni c_b ::= \textbf{assume}\ e_b \mid \textbf{assert}\ e_b \mid x := e_b \mid \textbf{havoc}\ x$ $BStmtBlock \ni b_b ::= \overrightarrow{c_b}; if_b$

$BIfOpt \ni if_b ::= \textbf{if}(e_b)\ \{s_b\}\ \textbf{else}\ \{s_b\} \mid \textbf{if}(*)\ \{s_b\}\ \textbf{else}\ \{s_b\} \mid \epsilon$ $BStmt \ni s_b ::= \overrightarrow{b_b}$

Fig. 1. The syntax of our formalised Viper subset (top, blue keywords) and corresponding Boogie subset (bottom, with subscript b, orange keywords) without top-level declarations. τ (τ_b), bop, and uop denote types, binary and unary operations, respectively.

> These generated proofs can be checked independently in Isabelle, which ensures front-end soundness of the Viper verifier.
> - Our evaluation on a diverse set of Viper programs demonstrates our approach's effectiveness: we were able to generate proofs and check them in Isabelle fully automatically in all cases.
> - As part of justifying the axioms used in Boogie programs, we provide the first approach to formally deal with a restricted version of Boogie's (impredicatively-)*polymorphic maps* [30].

Outline. Sec. 2 provides the necessary background on Viper and Boogie. Sec. 3 introduces our forward simulation methodology for relating Viper and Boogie statements. Sec. 4 presents how we formally validate the existing Viper-to-Boogie implementation using our forward simulation methodology. Sec. 5 evaluates the proofs generated by our instrumentation. Sec. 6 presents related work and Sec. 7 concludes. Our publicly-available artifact [37] contains the Isabelle formalisation for Sec. 2, Sec. 3, and Sec. 4, our proof-producing Viper-to-Boogie implementation, and the examples used for the evaluation. Further details are available in our technical report [38] (hereafter, TR).

2 VIPER AND BOOGIE: BACKGROUND AND SEMANTICS

In this section, we present the necessary background on the Viper and Boogie languages. We introduce our supported Viper subset and the corresponding Boogie subset targeted by the pre-existing Viper-to-Boogie implementation (Sec. 2.1), give an overview of the semantics of Boogie (Sec. 2.2) and Viper (Sec. 2.3), and finally show an example of the translation used by the Viper-to-Boogie implementation (Sec. 2.4).

2.1 The Viper and Boogie Languages

Viper programs in the subset considered here consist of a set of top-level declarations of fields (reference-field pairs are used to access the heap) and methods. Boogie programs consist of a set of top-level declarations of global variables, constants, uninterpreted (polymorphic) functions, type constructors, axioms (which constrain the constants and functions), and procedures. Both languages are imperative and separate *statements* from *expressions* (whose evaluation have no side-effects). Viper additionally has separate *assertions*. The body of each Viper method and Boogie procedure is a statement. Viper methods have pre- and post-conditions (assertions); method calls are verified modularly against these assertions.[1] In Viper, scoped variables can be declared within statements; Boogie procedures declare all variables upfront. Our supported Viper and Boogie statements, assertions, and expressions are shown in Fig. 1. Both languages have the same control flow elements

---

[1]Boogie supports pre-/post-conditions and procedure calls, but they are not used by the Viper-to-Boogie implementation.

and have some built-in types in common (e.g. Booleans and integers). Viper additionally provides a single *reference* type, and supports reading from and writing to heap locations via a field access e.f, where e is a reference expression and f a field.

Our validation generates proofs that connect the abstract syntax tree (AST) of a Viper program (as represented by the Viper verifier) with the AST of the corresponding Boogie program (as represented by the Boogie verifier).[2] Proof generation is complicated by the fact that the Viper and Boogie ASTs are structured differently. As shown in Fig. 1, the Viper AST uses a standard *sequential composition* $s_1; s_2$, whereas a Boogie statement is given by a list of *statement blocks*. Each statement block $\overrightarrow{c_b}; if_b$ consists of a list of *simple commands* (i.e. no control flow), followed by either an if-statement or an empty statement (ϵ).

As is typical for verifiers for higher-level languages, Viper's verification methodology employs a custom advanced program logic, in this case based on a flavour of separation logic (SL) called *implicit dynamic frames* (IDF) [36, 41] which reasons about the heap via *permissions*. Viper's assertions include the *accessibility predicate* **acc**(e.f,p), which represents a *resource* (a logical notion which can be neither freely fabricated nor duplicated): the fractional (p) amount of *permission to access heap location* e.f.[3] Fractional permission amounts [6] range between 0 and 1; nonzero permission is required to *read* heap locations and full (1) permission is required to *write to* heap locations. $A * B$ expresses the *separating conjunction* from SL, which specifies that the permissions in A and B must *sum up to an amount currently held*. One difference between IDF and SL is that IDF (and thus, Viper) supports general heap-dependent expressions such as x.val = 5 or x.f.f, whose evaluation is *partial* (only allowed with suitable permissions); this necessitates a notion of *well-definedness* checks on expressions (see Sec. 2.3). Boogie does not provide built-in heap reasoning, and uses a much simpler program logic: its assertions are (total) formulas in first-order logic.

The presence of a heap in Viper also results in a very different state model. A Viper state consists of a variable store, a heap (mapping heap locations to current values) and a *permission mask* (mapping heap locations to current permission amounts); a Boogie state is simply a variable store.

The main Viper features *not* included in our subset are loops, more-complex resource assertions (predicates, magic wands, iterated separating conjunctions), heap-dependent functions, and domains. Adding support for loops is straightforward: their semantics can be desugared via their invariant, in a pattern similar to method calls that we already support. For other features more work would be required, but we are confident that these extensions would fit within our general methodology.

2.2 Boogie Semantics

We extend our existing operational Boogie semantics formalised in Isabelle [39] to support the statements in Fig. 1, and reuse many components including the state model and the semantics of simple commands. The semantics of Boogie statements is expressed via program executions. A finite program execution has one of three outcomes: (1) it *fails*, because an **assert** e command is reached in a state that does not satisfy the Boolean expression e, (2) it *stops*, because an **assume** e command is reached in a state that does not satisfy the Boolean expression e, or (3) it *succeeds*, because neither of the first two situations occur. The three outcomes are represented formally via: (1) a failure outcome F, (2) a *magic* outcome M for when the execution stops, and (3) a *normal* outcome N(σ_b) in all other cases, where σ_b is the resulting Boogie state, which is given by a mapping from variables to values. Assignments and **havoc** commands always succeed; **havoc** x nondeterministically assigns a value of x's declared type to x.

---

[2]The Viper-to-Boogie implementation passes the Boogie program to the Boogie verifier via a text file. Targeting the Boogie AST as represented by the Boogie verifier in the proof avoids the need to trust the Boogie parser, and also generalises to verifier implementations that directly target the Boogie verifier's AST such as Dafny.

[3]For readers familiar with separation logics, this is analogous to a fractional points-to assertion in a separation logic.

Formally, executions of Boogie statements are expressed via a small-step semantics. The judgement $\Gamma_b \vdash (\gamma, N(\sigma_b)) \rightarrow_b^* (\gamma', r_b)$ expresses a finite execution w.r.t. *Boogie context* Γ_b that takes 0 or more steps starting from the *program point* γ and Boogie state σ_b, and ending in the program point γ' and outcome r_b. A Boogie context includes the interpretation of uninterpreted types and functions, and the types of declared variables. A program point is given by a pair of the currently active statement block b and the continuation representing the statement to be executed after b. A continuation is either the empty continuation (i.e. nothing to execute) or a sequential continuation (i.e. a statement block followed by a continuation). A continuation-based small-step semantics avoids the need for local search rules commonly required in a small-step semantics [1].

2.3 Viper Semantics

To our knowledge, there is no mechanised semantics for any fragment of the Viper language; we outline the main points of the one we have formalised here. We give a big-step operational semantics to Viper statements via program executions again with three possible outcomes for finite executions: *failure* F, *magic* M, and *normal outcomes* $N(\sigma_v)$ where σ_v is a *Viper state*. A Viper state σ_v comprises a local variable mapping $st(\sigma_v)$, a heap $h(\sigma_v)$ (a total mapping from heap locations to values), and a permission mask $\pi(\sigma_v)$ (a total mapping from heap locations to permission amounts). The judgement $\Gamma_v \vdash \langle s, \sigma_v \rangle \rightarrow_v r_v$ holds if in the *Viper context* Γ_v (fixing the declarations of methods, fields and local variables) the execution of statement s in the state σ_v terminates with outcome r_v. Determining the outcome of a Viper execution is more involved than for Boogie as we will see below for the **inhale** and **exhale** operations. Our semantics takes care that all Viper states are *consistent*, i.e. have *consistent permission masks* (mapping each location to values between 0 and 1); executions that would produce inconsistent states in this sense are pruned by going to M.

Formalising expression evaluation requires care for Viper, since, in a given state, not even all type-correct expressions are *well-defined*: in our subset this can be either because of (1) division by zero, or (2) dereferencing a heap location for which no permission is held (subsuming null dereferences). In our semantics, evaluating an ill-defined expression causes execution to fail (in contrast to Boogie, where expression evaluation cannot fail). Our judgement $\langle e, \sigma_v \rangle \Downarrow V(v)$ expresses that expression e evaluates to a value v in state σ_v (in particular, e is well-defined in σ_v) and $\langle e, \sigma_v \rangle \Downarrow \notdef$ expresses that e is ill-defined in σ_v.

Viper uses two main primitives to encode separation logic reasoning: (1) **inhale** A adds the permissions specified by assertion A to the state, and *stops* any execution where either a logical constraint in A does not hold (these are *assumed*) or the added permissions would yield an inconsistent state. (2) **exhale** A *removes* the permissions specified by A, and *fails* if either insufficient permissions are held or if a constraint in A does not hold; for any heap locations to which *all permission was removed*, an **exhale** also non-deterministically assigns arbitrary values.[4] This non-deterministic assignment reflects the fact that, while our Viper states employ total heaps (typical for IDF [36]), the values stored in heap locations without permission should be unconstrained.

inhale and **exhale** operations are typically used in Viper to encode external or more-complex operations [34]. For instance, a Viper method call is expressed by exhaling the precondition and then inhaling the postcondition of the callee; the nondeterministic assignments made by the **exhale** model possible side effects of the call. We present here some of the key rules for **exhale**, which will be used later in this paper. Additional rules for **inhale** are presented in the appendix (App. A of the TR [38]); the complete rules are included in our Isabelle formalisation.

---

[4]For separation-logic-versed readers, the Hoare triples $\{R\}$ **inhale** A $\{R * A\}$ and $\{R * A\}$ **exhale** A $\{R\}$ reflect this behavior (assuming the expressions in A and R are well-defined).

$$\frac{\begin{array}{c} \sigma_v \vdash \langle A, \sigma_v \rangle \rightarrow_{rc} \mathsf{N}(\sigma_v') \\ \mathrm{nonDet}(\sigma_v, \sigma_v', \sigma_v'') \end{array}}{\Gamma_v \vdash \langle \mathbf{exhale}\ A, \sigma_v \rangle \rightarrow_v \mathsf{N}(\sigma_v'')}\ (\textsc{exh-succ}) \qquad \frac{\sigma_v \vdash \langle A, \sigma_v \rangle \rightarrow_{rc} \mathsf{F}}{\Gamma_v \vdash \langle \mathbf{exhale}\ A, \sigma_v \rangle \rightarrow_v \mathsf{F}}\ (\textsc{exh-fail})$$

$$\frac{\begin{array}{c} \sigma_v^0 \vdash \langle A, \sigma_v \rangle \rightarrow_{rc} \mathsf{N}(\sigma_v') \\ \sigma_v^0 \vdash \langle B, \sigma_v' \rangle \rightarrow_{rc} r_v \end{array}}{\sigma_v^0 \vdash \langle A * B, \sigma_v \rangle \rightarrow_{rc} r_v}\ (\textsc{rc-sep}) \qquad \frac{\begin{array}{c} \langle e, \sigma_v^0 \rangle \Downarrow \mathsf{V}(r) \quad \langle e_p, \sigma_v^0 \rangle \Downarrow \mathsf{V}(p) \\ r_v = \mathit{if}\ \mathrm{exhAccSucc}(r, p, \sigma_v)\ \mathit{then}\ \mathsf{N}(\sigma_v^R)\ \mathit{else}\ \mathsf{F} \end{array}}{\sigma_v^0 \vdash \langle \mathbf{acc}(e.f, e_p), \sigma_v \rangle \rightarrow_{rc} r_v}\ (\textsc{rc-acc})$$

$$\mathrm{nonDet}(\sigma_v, \sigma_v', \sigma_v'') \triangleq \begin{array}{l} \mathrm{st}(\sigma_v'') = \mathrm{st}(\sigma_v') \wedge \pi(\sigma_v'') = \pi(\sigma_v') \wedge \\ \forall l.\ (\pi(\sigma_v)(l) = 0 \vee \pi(\sigma_v')(l) > 0) \Rightarrow h(\sigma_v'')(l) = h(\sigma_v')(l) \end{array}$$

$$\mathrm{exhAccSucc}(r, p, \sigma_v) \triangleq p \geq 0 \wedge (r = \mathbf{null}\ ?\ p = 0 : \pi(\sigma_v)(r.f) \geq p) \qquad \sigma_v^R \triangleq \mathrm{rem}(\sigma_v, r, f, p)$$

Fig. 2. A subset of the rules for the formal semantics of exhale. $\mathrm{rem}(\sigma_v, r, f, p)$ is the state σ_v where permission p is removed from $r.f$.

An **exhale** A must cause the loss of heap value information (via non-deterministic assignments) in general, but also needs to check that logical constraints *were* true when the exhale started. Our semantics for **exhale** A first removes the permissions and checks the constraints specified in A *without changing the heap yet* via an intermediate operation **remcheck** A; only then does it apply nondeterministic assignments. The inference rule EXH-SUCC in Fig. 2 formalises this behaviour for the case when **exhale** A succeeds. The big-step judgement $\sigma_v \vdash \langle A, \sigma_v \rangle \rightarrow_{rc} \mathsf{N}(\sigma_v')$ defines the successful execution of a **remcheck** A operation from σ_v to σ_v'. nonDet specifies the nondeterministic assignment for all heap locations for which **remcheck** A removed all permission. The case when **remcheck** A (and thus **exhale** A) fails, is captured by the rule EXH-FAIL.

Our semantics for **remcheck** A decomposes the assertion A from left to right: That is, **remcheck** $A * B$ first executes **remcheck** A and then **remcheck** B (rule RC-SEP formalises the case when **remcheck** A succeeds; if **remcheck** A fails, then **remcheck** $A * B$ also fails). However, we need to also take care that the removal of permissions on-the-fly doesn't cause subexpressions to be considered ill-defined, e.g. for the subexpression $x.f == 1$ in **remcheck** $\mathbf{acc}(x.f, 1) * x.f ==$ 1 which comes after the permission to $x.f$ is removed. Thus, our judgement carries both an *expression evaluation state* (σ_v^0 in RC-SEP) in which expressions are evaluated and a *reduction state* (σ_v and σ_v' in RC-SEP) from which permissions are removed. Rule RC-ACC for **remcheck** $\mathbf{acc}(e.f, e_p)$ models removing e_p permission for heap location $e.f$. The operation succeeds (expressed by $\mathrm{exhAccSucc}(r, p, \sigma_v)$) iff (1) the to-be-removed permission is nonnegative and, (2) there is sufficient permission. Rule RC-ACC is applicable only if e and e_p are well-defined; there is a separate rule (not shown here) expressing that **remcheck** $\mathbf{acc}(e.f, e_p)$ fails if e or e_p are ill-defined.

2.4 Example Viper-to-Boogie Translation

To give a flavour of a translation of a Viper statement into a Boogie statement, consider Fig. 3, which shows a simplified translation used by the existing Viper-to-Boogie implementation. The Viper statement first adds permission to $\mathtt{x.f}$, then updates $\mathtt{y.g}$, and finally removes the added permission to $\mathtt{x.f}$ and checks that $\mathtt{y.g}$ is greater than $\mathtt{x.f}$. This sequence of operations occurs, for instance, when verifying a method with the permission to $\mathtt{x.f}$ as precondition, the field update as method body, and the exhaled assertion as postcondition.

The corresponding Boogie program is significantly larger. The **inhale** is encoded on lines 1-4, the assignment is encoded on lines 5-7, and the **exhale** is encoded on lines 8-18. The Boogie

```
inhale acc(x.f, q)
y.g := x.f+1
exhale acc(x.f, q) * y.g > x.f
```

$\leadsto$

```
1   tmp := q; assert tmp >= 0;
2   assume tmp > 0 ==> x != null;
3   M[x,f] += tmp;
4   assume GoodMask(M);
5   assert M[x,f] > 0; assert M[y,g] == 1;
6   H[y,g] := H[x,f]+1;
7   assume GoodMask(M);
8   WM := M;
9   tmp := q; assert tmp >= 0;
10  if(tmp != 0) {
11    assert M[x,f] >= tmp;
12  }
13  M[x,f] -= tmp;
14  assert WM[y,g] > 0; assert WM[x,f] > 0;
15  assert H[y,g] > H[x,f];
16  havoc H'; assume idOnPositive(H,H',M);
17  H := H';
18  assume GoodMask(M);
```

Fig. 3. A Viper statement (on the left) and the corresponding (simplified) Boogie statement (on the right) that is emitted by the current Viper-to-Boogie implementation.

program uses map-typed variables H and M to model the heap and permissions, respectively.[5] The uninterpreted function GoodMask expresses when a permission mask is consistent; an axiom constrains the function correspondingly. The permission mask of the expression evaluation state during the **remcheck** operation is captured by the auxiliary variable WM (line 8). All locations in the assertion are checked to have positive permission w.r.t. WM. The corresponding nondeterministic assignment of heap values is performed on lines 16-17, where a heap H' is nondeterministically obtained via **havoc** H' and then constrained to match the original heap H on all locations where there is positive permission (w.r.t. M) via the **assume** statement; an axiom constrains the uninterpreted function idOnPositive correspondingly. Note that this Boogie encoding overapproximates the nondeterministic assignment specified by the Viper semantics: assigning new values to all locations without permission, rather than only those newly without permission. Even this tiny snippet of code illustrates the explosion in concerns, complexity and the inobvious mapping between concepts in one language and the other, all of which must be taken care of in a formal validation approach.

3 A FORWARD SIMULATION METHODOLOGY FOR FRONT-END TRANSLATIONS

A front-end translation is *sound* iff the correctness of an input program is implied by the correctness of the correspondingly-translated IVL program. In our setting: a Viper program (resp. a Boogie program) is *correct* if each of its methods (resp. procedures) is correct. At a high level (details in Sec. 4.5), a method (resp. procedure) is correct if its body has no failing executions. Thus, proving soundness of the Viper-to-Boogie translation boils down to proving that *if* the Viper program has a failing execution, then the translated Boogie program has one also.

We generate such proofs *automatically* via a novel general methodology for proving *forward simulations* [32] between source and IVL target statements. We observed early on that generating such proofs directly based on knowledge of the entire translation would require handling the entire

---

[5]The notation m[a] is syntactic sugar here. We describe in Sec. 5 how maps are represented using the subset from Fig. 1.

semantic gap between the source and target languages monolithically in one result, which would be both infeasible to automate effectively and highly-brittle to any changes in the translation.

Instead, our methodology employs a combination of key strategies that work together to achieve reliable and robust automation of our formal simulation results: (1) syntactic and semantic *decompositions* into smaller and more-focused simulation sub-results that are easier to automate, (2) *generic simulation judgements* which can be instantiated to obtain the diverse simulation notions we require, (3) *generic composition lemmas* which factor out common idioms arising in diverse facets of the translation, and (4) *contextual hypotheses* which can be injected into simulation proofs to handle non-locality of certain translation checks. We present these key ingredients of our methodology in this section. We illustrate them for Viper and Boogie, but they can be naturally ported to other front-end translations if one provides a formal semantics for the input language and IVL, because they are designed to abstract over states, relations and statements employed in a translation.

3.1 Focusing Forward Simulation Proofs by Decomposition

Intuitively, a forward simulation between a Viper and a Boogie statement shows that for any execution of the Viper statement, there exists a corresponding execution of the Boogie statement that *simulates* it. By defining the simulation such that a *failing* Viper execution is simulated only by *failing* Boogie executions, a forward simulation implies our desired result in particular.

To tackle the complexity of automatically (and reliably) generating simulation proofs in general for the Viper-to-Boogie translation, we employ a variety of strategies for aggressively decomposing the desired simulation result into smaller and simpler sub-goals that are themselves still simulation results. These decompositions are sometimes intuitive based on the syntax: for example, in the case of decomposing simulation of a Viper sequential composition into simulations for its constituent statements. However, we go *further than the syntax*, decomposing across different *semantic concerns* for the *same* Viper statement, into what we call Viper *effects*.

For example, we discussed in Sec. 2.3 that the semantics of `exhale` consists of two effects, `remcheck` and a nondeterministic assignment. The simulation proofs for each of these Viper effects are made separately, and then composed for a simulation proof for the primitive statement as a whole; this would in turn be composed with simulation proofs for other sequentially-composed statements, and so on. Note in particular, that simulation proofs may need to relate only a *part of* the semantics of a Viper statement to some appropriate Boogie code, a technicality which requires special care when tracking the *relations* between corresponding states in the two programs.

Via our decompositions, each resulting simulation proof focuses on a different specific semantic concern with respect to the translation in question; these proofs can be made simple enough to discharge automatically, optionally with tailored tactics. However without care, our decomposition approach could lead easily to an explosion of ad hoc simulation judgements with disparate forms and parameters. Instead, our simulation methodology defines a *single, generic* simulation judgement which can be instantiated appropriately to define each particular simulation judgement required. We design our generic judgements to support instantiations which reflect not only the semantics of the particular effect in isolation, but to optionally include additional contextual information to be propagated to specialise and aid the simulation proof itself.

3.2 One Simulation Judgement to Rule Them All

Our generic forward simulation judgement sim is defined in Fig. 4. All concrete forward simulations (e.g. for statements, well-definedness checks, etc.) are instantiations of this judgement. As well as aiding understanding, this approach enables both tactics which manipulate this generic judgement directly, and *generic composition proof rules* which embody recurring proof idioms in a way which is again parametric with the specific simulations in question (Sec. 3.3).

$$\text{sim}_{\Gamma_b}(R_{in}, R_{out}, Succ, Fail, \gamma_{in}, \gamma_{out}) \triangleq \forall \tau, \sigma_b.\ R_{in}(\tau, \sigma_b) \implies$$

$$(\forall \tau'.\ Succ(\tau, \tau') \implies \exists \sigma_b'.\ \Gamma_b \vdash (\gamma_{in}, \mathsf{N}(\sigma_b)) \to_b^* (\gamma_{out}, \mathsf{N}(\sigma_b')) \wedge R_{out}(\tau', \sigma_b')) \wedge \quad \text{(Success case)}$$

$$(Fail(\tau) \implies \exists \gamma'.\ \Gamma_b \vdash (\gamma_{in}, \sigma_b) \to_b^* (\gamma', \mathsf{F})) \quad\quad\quad\quad\quad\quad\quad \text{(Failure case)}$$

$$\text{stmSim}_{\Gamma_v, \Gamma_b}(R, R', s, \gamma, \gamma') \triangleq$$

$$\text{sim}_{\Gamma_b}(R, R', \lambda \sigma_v\ \sigma_v'.\ \Gamma_v \vdash \langle s, \sigma_v \rangle \to_v \mathsf{N}(\sigma_v'), \lambda \sigma_v.\ \Gamma_v \vdash \langle s, \sigma_v \rangle \to_v \mathsf{F}, \gamma, \gamma')$$

$$\text{wfSim}_{\Gamma_b}(R, R', es, \gamma, \gamma') \triangleq \text{sim}_{\Gamma_b} \left(\begin{array}{l} R, R', (\lambda \sigma_v\ \sigma_v'.\ \sigma_v = \sigma_v' \wedge \exists vs.\ \langle es, \sigma_v \rangle [\Downarrow] \mathsf{V}(vs)), \\ (\lambda \sigma_v.\ \langle es, \sigma_v \rangle [\Downarrow] \not{\xi}), \gamma, \gamma' \end{array} \right)$$

$$\text{rcSim}_{\Gamma_b}(R, R', A, \gamma, \gamma') \triangleq \text{sim}_{\Gamma_b} \left(\begin{array}{l} R, R', (\lambda(\sigma_v^0, \sigma_v)\ (\sigma_v^1, \sigma_v').\ \sigma_v^0 = \sigma_v^1 \wedge \sigma_v^0 \vdash \langle A, \sigma_v \rangle \to_{rc} \mathsf{N}(\sigma_v')), \\ (\lambda(\sigma_v^0, \sigma_v).\ \sigma_v^0 \vdash \langle A, \sigma_v \rangle \to_{rc} \mathsf{F}), \gamma, \gamma' \end{array} \right)$$

Fig. 4. The definition of the generic forward simulation judgement and three common instantiations. The judgement $\langle es, \sigma_v \rangle [\Downarrow] r$ lifts the evaluation of an expression (see Sec. 2.3) to a list of expressions es.

sim is defined in terms of multiple parameters: (1) the Boogie context Γ_b, (2) an *input relation* R_{in} and an *output relation* R_{out} on Viper and Boogie states, (3) a *success predicate Succ* characterising the set of input and output Viper state pairs (τ, τ') for which there is a successful Viper execution from τ to τ', (4) a *failure predicate Fail* characterising the set of input Viper states that result in a failing execution, (5) input and output Boogie program points γ_{in} and γ_{out} where the Boogie executions are expected to start and end, respectively. The success and failure predicate together abstractly describe the set of Viper executions that must be shown to be simulated.

$\text{sim}_{\Gamma_b}(R_{in}, R_{out}, Succ, Fail, \gamma_{in}, \gamma_{out})$ holds iff for any Viper and Boogie input states related by R_{in}, the following two conditions hold: (1) for any successful Viper execution from the input Viper state to an output Viper state τ', there must be a Boogie execution from program point γ_{in} and the input Boogie state to program point γ_{out} and some output Boogie state that is related to τ' by R_{out}, and (2) if the Viper execution fails in the input state, then there must be a failing Boogie execution from γ_{in} and the input Boogie state (the reached Boogie program point need not be γ_{out}). The second condition is the end goal that we need to show soundness of the Viper-to-Boogie translation. The first condition is needed in order to derive sim compositionally; it guarantees, for example, that not all Boogie executions for a successful Viper execution produce a magic outcome.

Three important instantiations of sim that we use are shown at the bottom of Fig. 4. stmSim is the forward simulation for Viper statements, where the success and failure predicates are instantiated to be a successful and a failing Viper statement reduction, respectively. Thus, the resulting failure case in sim directly gives us the key property to show the soundness of a Viper-to-Boogie translation. wfSim is the forward simulation for the well-definedness check of a list of Viper expressions. Here, the instantiation of the success predicate explicitly expresses that the Viper state does not change during the evaluation of expressions. rcSim is the forward simulation for **remcheck**. Here, the instantiation makes use of the fact that the generic simulation judgement sim is in fact also (implicitly, here) parametric with the notions of states employed: the "Viper state" is in fact instantiated to be a pair of standard Viper states in this case, where the first Viper state represents the expression evaluation state and the second Viper state represents the reduction state (see Sec. 2.3 for this distinction). The success predicate expresses that the expression evaluation state does not change during a **remcheck** operation. These three common instantiations are all expressed directly via the Viper reduction judgements introduced in Sec. 2.3. Like the generic simulation judgement, the three instantiations *are themselves generic*, abstracting away *how* the Viper and Boogie states

$$\frac{\begin{array}{c} \mathsf{sim}_{\Gamma_b}(R, R', S_1, F_1, \gamma, \gamma') \\ \mathsf{sim}_{\Gamma_b}(R', R'', S_2, F_2, \gamma', \gamma'') \\ \forall \tau, \tau''.\ S(\tau, \tau'') \Rightarrow \exists \tau'.\ S_1(\tau, \tau') \wedge S_2(\tau', \tau'') \\ \forall \tau.\ F(\tau) \Rightarrow F_1(\tau) \vee \exists \tau'.\ S_1(\tau, \tau') \wedge F_2(\tau') \end{array}}{\mathsf{sim}_{\Gamma_b}(R, R'', S, F, \gamma, \gamma'')}\ (\textsc{comp}) \qquad \frac{\begin{array}{c} \mathsf{bSim}_{\Gamma_b}(R, R_1, \gamma, \gamma_1) \\ \mathsf{sim}_{\Gamma_b}(R_1, R_2, S, F, \gamma_1, \gamma_2) \\ \mathsf{bSim}_{\Gamma_b}(R_2, R', \gamma_2, \gamma') \end{array}}{\mathsf{sim}_{\Gamma_b}(R, R', S, F, \gamma, \gamma')}\ (\textsc{bprop})$$

$$\frac{\begin{array}{c} \mathsf{stmSim}_{\Gamma_v, \Gamma_b}(R, R', s_1, \gamma, \gamma') \\ \mathsf{stmSim}_{\Gamma_v, \Gamma_b}(R', R'', s_2, \gamma', \gamma'') \end{array}}{\mathsf{stmSim}_{\Gamma_v, \Gamma_b}(R, R'', (s_1; s_2), \gamma, \gamma'')}\ (\textsc{seq-sim}) \qquad where \quad \begin{array}{c} \mathsf{bSim}_{\Gamma_b}(R, R', \gamma, \gamma') \triangleq \\ \mathsf{sim}_{\Gamma_b}(R, R', \lambda \tau\ \tau'.\ \tau = \tau', \lambda\_.\ \bot, \gamma, \gamma') \end{array}$$

Fig. 5. The instantiation-independent rules comp and bprop and the concrete rule for the simulation of $s_1; s_2$.

are related by taking the input and output state relations as parameters. As we will show in Sec. 3.4, we also use instantiations that do not just use Viper reduction judgements (e.g. to express the non-deterministic assignment of heap values in **remcheck**).

3.3 Instantiation-Independent Rules

Many simulation idioms arise repeatedly in a complex translation. Notions of sequential composition, conditional evaluation, stuttering steps are all good examples, which require a certain stylised formal justification to reason about. Our generic simulation judgement allows us to identify and formalise these idioms once and for all, providing, for example, generic composition lemmas that can be proved once and instantiated for different purposes. In this subsection, we present these idioms as inference rules, but in our formalisation they are expressed and proved as regular lemmas.

For example, we prove a single general composition rule from which we derive concrete rules to combine (1) simulations of s_1 and s_2 to a simulation of $s_1; s_2$, (2) simulations of **remcheck** A_1 and **remcheck** A_2 to **remcheck** $A_1 * A_2$, (3) simulations of **inhale** A_1 and **inhale** A_2 to **inhale** $A_1 * A_2$. The general composition rule comp in Fig. 5 captures the composition of two, possibly different, instantiations of sim, where the output relation and Boogie program point of the first instantiation match the input relation and program point of the second one. The two final premises constrain the resulting success and failure predicates. In particular, the composed Viper execution should fail only if either the first instantiation fails or if the second instantiation fails in a state successfully reached by the first one. The rule seq-sim in Fig. 5 shows the concrete composition rule for $s_1; s_2$, which is derived from comp. Note that seq-sim does not impose any constraints on the Boogie program points, which is crucial to handle Viper's and Boogie's disparate ASTs (see Sec. 2.1). We will discuss in Sec. 4.3 how we deal with the AST mismatch when automating proofs.

As a second example, the notion of simulation *stuttering steps* also arises in many ways, whenever some auxiliary Boogie code is generated that does not fully correspond to a step in the Viper source. This includes initialisations of auxiliary variables, or Boogie **assume** statements for properties from the current simulation state relation. This idiom is captured by the *Boogie propagation rule* bprop in Fig. 5, in which bSim expresses simulations in which the Viper state remains unchanged, and thus only the Boogie state may change (causing adjustment to the state relations).

3.4 Examples: Generic Decomposition in Action

As outlined above, the general strategy for our simulation methodology is to decompose our simulation goals as far as possible, while leaving as many parameters generic as we can to enable maximal reuse of our results and composition lemmas. While decomposition handles the semantic

$$\frac{\begin{array}{c} \text{rcSim}_{\Gamma_b}([\lambda(\sigma_v^0, \sigma_v)\ \sigma_b.\ \sigma_v^0 = \sigma_v \wedge R(\sigma_v, \sigma_b)], R', A, \gamma, \gamma') \quad \text{(sim. of } \textbf{remcheck } A) \\ \text{sim}_{\Gamma_b}(R', [\lambda(\_, \sigma_v)\ \sigma_b.\ R''(\sigma_v, \sigma_b)], Succ_2, \lambda\_.\ \bot, \gamma', \gamma'') \quad \text{(non-det. selection)} \end{array}}{\text{stmSim}_{\Gamma_v, \Gamma_b}(R, R'', \textbf{exhale } A, \gamma, \gamma'')} \text{(EXH-SIM)}$$

$$Succ_2 \triangleq \lambda(\sigma_v^0, \sigma_v)\ (\_, \sigma_v').\ \text{nonDet}(\sigma_v^0, \sigma_v, \sigma_v') \wedge \sigma_v^0 \vdash \langle A, \sigma_v^0 \rangle \rightarrow_{\text{rc}} \sigma_v$$

Fig. 6. Rule for the simulation of **exhale** A. The definition of nonDet is given in Fig. 2.

gap, our use of generic parameterisation provides the abstraction to address the diverse translations used in practical translational verifiers. In the following, we showcase our methodology on one rule, but the same ideas apply to all our formal rules (see a second example in App. B of the TR [38]).

Consider the rule EXH-SIM for the simulation of **exhale** A in Fig. 6. The first premise is expressed as a simulation of the first effect, **remcheck**, which we can express via the rcSim instantiation (see Fig. 4). The second premise models nondeterministic assignment, which is captured by the first conjunct nonDet of the corresponding success predicate and by the failure predicate, which reflects that the nondeterministic assignment cannot fail.

By modularly abstracting over the details of these premises, and the precise definitions of the states and state relations (e.g. the intermediate relation R' in this rule), we obtain robustness to diverse translations: our rules do not constrain *which exact Boogie statements* correspond to a Viper effect. For example, the Viper-to-Boogie implementation establishes the nondeterministic heap assignment premise in EXH-SIM in two different ways depending on whether the assertion contains an accessibility predicate **acc**$(e.f, p)$ or not; if not, then the implementation does not emit any Boogie code for the nondeterministic assignment, which is sound, since no permission is removed.

Note that this genericity does not prevent the rule from exploiting contextual information. For example, the input state relation of the first premise specifies that at the beginning of the **remcheck** A effect the expression evaluation state and the reduction state are the same. This property does not hold in general for executions of **remcheck** (e.g. it might not hold when executing the second conjunct of a separating conjunction), but it does hold here, at the beginning of an **exhale**. The second premise's success predicate includes the fact that the current Viper state was reached via **remcheck** A. This allows us, for example, to conclude that the nondeterministic assignment has no effect if **remcheck** A removes no permissions, which is required to justify the case when the implementation does not emit Boogie code for the nondeterministic assignment.

3.5 Injecting Non-Local Hypotheses into Simulation Proofs

Our rules are designed to be parametric in the state relation between the Viper and Boogie state and permit adjusting this state relation at different points in the simulation proof (e.g. via the Boogie propagation rule BPROP in Fig. 5). In principle, this allows the injection of arbitrary non-locally-justified hypotheses into all of our simulation judgements. However, automating the *usage* of general logical assumptions embedded into our state relations can become a challenge in itself.

For example, the Viper-to-Boogie implementation omits the well-definedness checks of expressions in the translation of **remcheck** A and **inhale** A in certain cases (as we will discuss in Sec. 4.2). This is justified, because A is checked to be *well-formed* non-locally in those cases, but to *use* this additional hypothesis requires propagating and adjusting it through the cases of the definition of **remcheck** A and **inhale** A.

As a final ingredient of our methodology, to avoid these recurring adaptations and proof steps, we allow *specialised* instantiations of the generic forward simulation judgement sim that encapsulate

$$\frac{\begin{array}{c} \text{rcInvSim}_{\Gamma_b}^Q(R, R', A_1, \gamma, \gamma') \quad \text{rcInvSim}_{\Gamma_b}^Q(R', R'', A_2, \gamma', \gamma'') \\ \forall \sigma_v^0, \sigma_v.\ Q(A_1 * A_2, (\sigma_v^0, \sigma_v)) \Rightarrow \left(\begin{array}{l} Q(A_1, (\sigma_v^0, \sigma_v)) \land \\ \forall \sigma_v'.\ \sigma_v^0 \vdash \langle A_1, \sigma_v \rangle \rightarrow_{\text{rc}} \mathsf{N}(\sigma_v') \Rightarrow Q(A_2, (\sigma_v^0, \sigma_v')) \end{array} \right) \end{array}}{\text{rcInvSim}_{\Gamma_b}^Q(R, R'', (A_1 * A_2), \gamma, \gamma'')} \ (\text{RSEP-SIM})$$

$$\text{rcInvSim}_{\Gamma_b}^Q(R, R', A, \gamma, \gamma') \triangleq \text{rcSim}_{\Gamma_b}((\lambda \tau, \sigma_b.\ R(\tau, \sigma_b) \land Q(A, \tau)), R', A, \gamma, \gamma')$$

Fig. 7. The instantiation for simulating `remcheck` A with assertion predicate Q (bottom) and the corresponding rule for the separating conjunction (top).

these extra hypotheses as *additional* premises. Thus, applications of the rule can work with a fixed state relation and replace recurring proof steps by the justification of an additional premise.

For example, Fig. 7 shows (at the bottom) an instantiation of sim that expresses the simulation of `remcheck` A, parameterised with a predicate Q on assertions. Its definition in terms of rcSim requires $Q(A, \tau)$ to hold as part of the input state relation. The specialised rule RSEP-SIM (top of Fig. 7) for `remcheck` $A_1 * A_2$ decomposes the simulation into simulations for A_1 and A_2.[6] Both sub-simulations use the *same* predicate Q, such that applications of the rule do *not* need to adjust the state relations explicitly to reflect that, for example, Q holds for A_1 and A_2 in the respective states. This property is ensured by the third premise. In practice, for a specific Q, we prove the third premise once and for all for all assertions A_1 and A_2, which avoids the recurring proof steps that would be necessary without the specialised rule. Note that the same parameter can be instantiated in many ways to capture different non-local hypotheses for different applications of the same rule.

In summary, our methodology solves all three challenges outlined in the introduction. The *large semantic gap* between the input language and the IVL is handled by decomposing the statements of the input language into smaller effects and defining for each of them instantiations of a generic forward simulation relation. The parameterisation of this relation allows us, in particular, to capture information about the context in which the effects are executed. This parameterisation also supports *diverse translations* by abstracting from the details of the translation. Finally, *non-locality* is handled by capturing properties checked elsewhere in the state relations, and by devising specialised rules that simplify the proof generation. All of these ideas are needed to validate the existing Viper-to-Boogie translation, but apply equally to other front-end translations.

4 PUTTING THE METHODOLOGY TO WORK

This section presents ideas for applying the methodology from Sec. 3 to concrete front-end translations. In particular, the section presents our instantiation of the state relation (Sec. 4.1), a concrete instance of non-local reasoning (Sec. 4.2), and how our proof automation works (Sec. 4.3). Finally, the section discusses the background theory for Boogie (Sec. 4.4), which includes polymorphic maps, and shows how to use forward simulation proofs to generate the final theorem (Sec. 4.5).

4.1 State Relation

Our rules for deriving forward simulation judgements (Sec. 3) allow us to adjust state relations as needed during a simulation proof. We use this flexibility in many ways, e.g. when (1) a scoped Viper variable is introduced, (2) a new auxiliary Boogie variable is introduced, (3) the Boogie variables

---

[6]RSEP-SIM can be derived from the instantiation-independent composition rule (Fig. 5) and consequence rule CONS (App. C of the TR [38]).

tracking the Viper state are changed. To facilitate proof automation for handling such adjustments, we build in a stylised form for expressing state relations for this translation via two parameters: a partial *auxiliary variable map* from auxiliary Boogie variables to *logical conditions they each satisfy*, and a *translation record* specifying how key Viper components are represented in the Boogie state; the scenarios above are all handled by adjusting one of these two parameters. Translation records comprise: (1) a mapping $var(Tr)$ from Viper variables to their Boogie counterparts, (2) the Boogie variables representing the Viper heap $H(Tr)$ and permission mask $M(Tr)$ (and whenever we use a separate expression evaluation state, the corresponding variables representing the heap $H^0(Tr)$ and mask $M^0(Tr)$) and (3) a mapping $field(Tr)$ from Viper fields to corresponding Boogie constants.

The following definition shows a simplified excerpt of our state relation instantiation SR for translation record Tr and auxiliary variable map AV, where σ_v and σ_b are the Viper and Boogie states, and σ_v^0 is a distinguished Viper expression evaluation state (if there is none, then $\sigma_v = \sigma_v^0$):

$$\mathsf{SR}_{\Gamma_b}^{Tr,AV}((\sigma_v^0, \sigma_v), \sigma_b) \triangleq \mathsf{consistent}(\sigma_v^0) \wedge \mathsf{consistent}(\sigma_v) \wedge$$

$$\mathsf{fieldRel}_{\Gamma_b}(field(Tr), \sigma_b) \wedge (\forall x, P.\ AV(x) = P \Rightarrow P(\sigma_b(x))) \wedge$$

$$\mathsf{stRel}_{\Gamma_b}(var(Tr), \sigma_v, \sigma_b) \wedge \mathsf{hmRel}_{\Gamma_b}(H(Tr), M(Tr), \sigma_v, \sigma_b) \wedge \mathsf{hmRel}_{\Gamma_b}(H^0(Tr), M^0(Tr), \sigma_v^0, \sigma_b) \wedge \dots$$

The first line ensures that the Viper states are consistent. The second line ensures that the Viper fields are represented in the Boogie state (fieldRel) and that for each (x, P) in the auxiliary variable map, P holds for the value of x. The third line ensures that the Boogie state correctly captures the Viper state: both in terms of its variable store (stRel) and heap and permission mask (hmRel).

4.2 Non-Locality

For most occurrences of `remcheck` A, the Viper-to-Boogie implementation generates well-definedness checks in the Boogie program corresponding to expressions evaluated in A. However, specifically when executing the `exhale` of a method call's precondition, the translation omits these well-definedness checks for the corresponding `remcheck` operation. This is justified by a *non-local check*: the Boogie code for the *callee's* translation checks that the callee's specification is *well-formed*, which implies that expressions evaluated in the precondition will always be well-defined.[7]

Given Viper's semantics, our standard simulation proof for `remcheck` A would fail if we did not reflect the consequences of this non-local guarantee *in a way that is used automatically during the proof*. We instantiate the general strategy outlined in Sec. 3.5 for this purpose, which allows us to choose a predicate Q_{pre} on assertions that will be applied throughout the simulation proof for `remcheck` A. Our strategy requires us to find Q_{pre} such that (a) it is implied by the non-local check elsewhere, and (b) it can be propagated identically to sub-assertions of A during the proof (e.g. satisfying the third premise of RSEP-SIM in Fig. 7, and similarly for other connectives).

In this case, we instantiate the predicate in our strategy with the following definition:

$$Q_{pre}(A, \sigma_v^0, \sigma_v) \triangleq \mathsf{consistent}(\sigma_v^0) \wedge \exists \sigma_v^i.\ \sigma_v \oplus \sigma_v^i \preceq \sigma_v^0 \wedge \neg \langle A, \sigma_v^i \rangle \rightarrow_{\mathsf{inh}} \mathsf{F}$$

Here, $\langle A, \sigma_v \rangle \rightarrow_{\mathsf{inh}} r_v$ holds iff $\Gamma_v \vdash \langle \mathbf{inhale}\ A, \sigma_v \rangle \rightarrow_v r_v$, and $\oplus$ and $\preceq$ (and later, $\ominus$) have standard pointwise meanings on permission masks, leaving heaps and stores identical. This predicate expresses that possibly after restoring some permissions (in σ_v^i) that we *had* at the start of the `exhale`, at least an `inhale` of A would not fail (i.e. expressions evaluated within A will be well-defined). The non-local check of the method precondition, which effectively checks that an `inhale` would not fail starting from an *empty* state (i.e. no permissions), implies the predicate for an empty σ_v^i. Showing formally that Q_{pre} can be propagated over connectives occurring in A requires in particular a technical lemma stating a partial *inversion* property between `remcheck` and `inhale`:

---

[7]There is an analogous non-local check for m's postcondition that we do not discuss here for simplicity of presentation.

Proof Tree T :

$$
\dfrac{
\boxed{\text{Proof } \mathcal{P}_2 \text{ (hint 3)}} \quad\quad \boxed{\text{Proof } \mathcal{P}_3 \text{ (no hint)}}
}{
\text{rcSim}_{\Gamma_b}(R_2, R_2, A_1 * A_2, \gamma_1, \gamma_3)
}
$$

$$
\dfrac{\text{rcSim}_{\Gamma_b}(R_2, R_2, A_1, \gamma_1, \gamma_2) \quad\quad \text{rcSim}_{\Gamma_b}(R_2, R_2, A_2, \gamma_2, \gamma_3)}{} \ (\textsc{rsep2-sim})
$$

$$
\dfrac{
\dfrac{\boxed{\text{Proof } \mathcal{P}_1 \text{ (hint 2)}}}{\text{bSim}_{\Gamma_b}(R_1, R_2, \gamma, \gamma_1) \quad\quad (\text{Proof Tree } T)} \ (\textsc{rcprop}) \quad\quad \dfrac{\boxed{\text{Proof } \mathcal{P}_4 \text{ (hints 4 and 5)}}}{\text{sim}_{\Gamma_b}(R_2, R_3, Succ_2, \lambda\_. \perp, \gamma_3, \gamma')} \ (\textsc{exh-sim})
}{
\text{stmSim}_{\Gamma_v, \Gamma_b}(R, R, \textbf{exhale } A_1 * A_2, \gamma, \gamma')
}
$$

$$
\dfrac{\text{rcSim}_{\Gamma_b}(R_1, R_2, A_1 * A_2, \gamma, \gamma_3)}{}
$$

hint 1

$A_1 \triangleq \textbf{acc}(\text{x.f}, \text{q}) \ A_2 \triangleq \text{y.g} > \text{x.f} \ \ R \triangleq \lambda(\sigma_v, \sigma_b). \ \text{SR}^{Tr,AV}_{\Gamma_b}((\sigma_v, \sigma_v), \sigma_b) \ R_3 \triangleq \lambda(\_, \sigma_v) \ \sigma_b. \ R(\sigma_v, \sigma_b)$

$R_1 \triangleq \lambda(\sigma_v^0, \sigma_v) \ \sigma_b. \ \sigma_v^0 = \sigma_v \wedge R(\sigma_v, \sigma_b) \quad R_2 \triangleq \text{SR}^{Tr_1, AV}_{\Gamma_b} \quad Tr_1 \triangleq Tr(M^0 \mapsto \text{WM})$

Fig. 8. Proof tree constructed by our proof automation for the simulation of **exhale acc**(x.f, q) * y.g > x.f via the Boogie statement in Fig. 3 on lines 8-18. The automation uses generated hints for the application of rule EXH-SIM, and for proofs at the leaves ($\mathcal{P}_i$; left abstract here). The Boogie program points γ, γ_1, γ_2, γ_3, and γ' are the points in Fig. 3 starting on lines 8, 9, 14, 16, and 18, respectively. SR is our state relation instantiation introduced in Sec. 4.1. $Succ_2$ is defined in Fig. 6 (where the assertion is $A_1 * A_2$). Rules RCPROP and RSEP2-SIM are derived from BPROP (Fig. 5) and RSEP-SIM (Fig. 7), respectively.

LEMMA 4.1. *Let A be an assertion and σ_v^0, σ_v', σ_v^i, σ_v^s be Viper states, where $\sigma_v^s = \sigma_v^i \oplus (\sigma_v \ominus \sigma_v')$ and σ_v^s is consistent. If $\sigma_v^0 \vdash \langle A, \sigma_v \rangle \rightarrow_{\text{rc}} N(\sigma_v')$ and $\neg \langle A, \sigma_v^i \rangle \rightarrow_{\text{inh}} F$ holds, then $\langle A, \sigma_v^i \rangle \rightarrow_{\text{inh}} N(\sigma_v^s)$.*

We prove this result by induction on the reduction of **remcheck**. The lemma essentially states that the permissions that get removed by **remcheck** A (expressed by $\sigma_v \ominus \sigma_v'$) are exactly those that will be added by a corresponding (non-failing) **inhale** A operation.

4.3 Proof Automation

We have extended the Viper-to-Boogie implementation to automatically generate an Isabelle proof relating the Viper and Boogie programs. To make this automatic generation possible, we instrument less than 500 lines of the existing implementation to produce *hints*, which provide extra information about the Boogie encoding. A core component of our proof automation is an Isabelle tactic that uses these hints to automatically prove forward simulations. The tactic applies the rules provided by our methodology (Sec. 3) to decompose simulations into smaller ones and generates proofs for *atomic simulations* that are not further decomposed. Our instrumentation generates two kinds of hints for the tactic: (1) hints indicating which candidate of multiple diverse translations is used, and (2) hints specifying how to instantiate parameters and discharge premises of a rule.

As a concrete example, consider Fig. 8, which shows the proof generated by our tactic (represented via a proof tree) for the forward simulation of **exhale acc**(x.f, q) * y.g > x.f via the Boogie statement in Fig. 3 on lines 8-18. Hints 1 and 4 in Fig. 8 are hints of the first kind. Hint 1 specifies that well-definedness checks are not omitted in the translation of **remcheck**; as a result, the tactic applies EXH-SIM, which does not track a separate predicate Q on assertions for the **remcheck** simulation (see Sec. 3.5). Hint 4 specifies that the nondeterministic heap assignment is not omitted in the Boogie code (see Sec. 3.4 for when it is omitted), which directs the tactic to use a specific rule (not shown in the figure). Hints 2, 3, and 5 in Fig. 8 are hints of the second kind. Each of them provides information on temporary Boogie variables (name and lemma showing the declared type is the

expected one) in Fig. 3. The temporary variables here are (1) WM to set up the expression evaluation state on line 8 (hint 2), which results in a change of the translation record in R_2 (see Sec. 4.1), (2) tmp to store the permission on line 9 (hint 3), which is used to adjust the auxiliary variable map (see Sec. 4.1) in proof $\mathcal{P}_2$, and (3) H' to perform the nondeterministic selection on line 16 (hint 5).

After decomposing the simulation, our tactic must automatically prove the atomic simulations. In Fig. 8, $\mathcal{P}_1$ and $\mathcal{P}_4$ are such proofs. $\mathcal{P}_2$ and $\mathcal{P}_3$ further decompose the simulation before reaching atomic simulations ($\mathcal{P}_2$ does so via the rule shown in App. B of the TR [38]). We use two main automation approaches for atomic simulations. Firstly, we prove (once and for all) simple lemmas about the behaviours of small sequences of simple Boogie commands; these are applied (and their hypotheses discharged) automatically when needed. These lemmas are used for only small parts of the overall translation. Secondly, we prove (once and for all) lemmas that capture effects simulated by **assume** and **assert** statements for arbitrary expressions. This generality enables a tactic to automatically prove Viper effects that are simulated via a combination of these two statements.

Our tactic uses both of these approaches for the example in Fig. 8. Proof $\mathcal{P}_2$ uses the second approach for justifying the nonfailure check for **remcheck acc**$(e.f, p)$ shown on lines 9-12 in Fig. 3.[8] Proofs $\mathcal{P}_1$ and $\mathcal{P}_4$ use the first approach. As part of proof $\mathcal{P}_4$, we use a lemma of the following form proved once and for all ($\mathcal{K}$ is a continuation and the free variables are universally quantified):

LEMMA 4.2. *If (1)* $SR_{\Gamma_b}^{Tr,AV}((\sigma_v, \sigma_v), \sigma_b)$, *(2)* $nonDet(\sigma_v^0, \sigma_v, \sigma_v')$, *(3)* $h = H(Tr) \wedge m = M(Tr)$, *and (4) … then there is a state* σ_b' *such that* $\Gamma_b \vdash ((\textbf{\textit{havoc }} h' :: \textbf{\textit{assume }} f(h, h', m) :: h \coloneqq h' :: \vec{c}; if, \mathcal{K}), N(\sigma_b)) \rightarrow_b^* ((\vec{c}; if, \mathcal{K}), N(\sigma_b'))$ *and* $SR_{\Gamma_b}^{Tr,AV}((\sigma_v', \sigma_v'), \sigma_b')$.

This lemma captures that a **havoc**-**assume**-assignment sequence simulates the nondeterministic heap assignment w.r.t. our state relation instantiation (see Sec. 4.1). The fourth premise (not shown here) includes constraints on f's interpretation and on h'.[9]

A general challenge when the tactic applies the rules from Sec. 3 is that the Viper and Boogie ASTs are structured differently (see Sec. 2.1). Thus, the automatic selection of Boogie program points in the premises of rules is not immediate. For example, when applying rule EXH-SIM in Fig. 8, the tactic cannot easily choose the intermediate program point γ_3 by inspecting the initial program point γ. Instead, the tactic starts proving the first premise with an *existentially quantified* γ_3. Once the proof reaches the goal of proof $\mathcal{P}_1$ (i.e. the first atomic simulation), it becomes clear how to advance the program point γ and, by the end of the proof of the first premise of EXH-SIM, the choice of γ_3 becomes clear. This strategy is enabled by our routine use of *schematic variables* in Isabelle (*evars* in other tools), for postponing the choice of witnesses for existentially-quantified values.

4.4 Background Theory and Polymorphic Maps

Boogie does not have any notion of a heap location or a Viper state. Such Viper (and other front-end) constructs are translated using particular global declarations in Boogie. A subset of the Boogie declarations always emitted by the Viper-to-Boogie implementation is given by:

- Uninterpreted types bref and bfield to model references and fields. bfield takes one type argument indicating the type of the corresponding Viper field.[10]
- An uninterpreted function GoodMask that maps a permission map to a Boolean and an axiom restricting this function to return true only if the permission map models a consistent Viper permission mask.

---

[8]The approach is designed to work without any changes to the tactic even if the expressions in the two **assert** statements were changed to be syntactically different.

[9]If the implementation changed the translation for the nondeterministic heap assignment, then we would have to adjust only the tactic's proof strategy for this assignment via a new lemma (i.e. proof $\mathcal{P}_4$ in Fig. 8); the rest remains unchanged.

[10]In practice, bfield takes one more type argument that we ignore for the sake of presentation.

$$\text{Correct}_b^G(p) \triangleq \forall \mathcal{T}, \mathcal{F}, \sigma_b. \ [\text{DeclsWf}_{G,p}(\mathcal{T}, \mathcal{F}) \wedge \text{AxiomSat}_G(\mathcal{T}, \mathcal{F}, \sigma_b)] \Longrightarrow$$

$$\forall \gamma', r_b'. \ \text{initCtxt}_b^G(p, \mathcal{T}, \mathcal{F}) \vdash (\text{init}_b(p), \mathsf{N}(\sigma_b)) \rightarrow_b^* (\gamma', r_b') \Longrightarrow r_b' \neq \mathsf{F}$$

$$\text{Correct}_v^{F,M}(m) \triangleq$$
$$\quad \forall \sigma_v. \ (\forall l.\pi(\sigma_v)(l) = 0) \Longrightarrow$$
$$\quad \quad \forall r_v. \ \text{initCtxt}_v^{M,F}(m) \vdash \langle \textbf{inhale} \ \text{pre}(m); \text{body}(m); \textbf{exhale} \ \text{post}(m), \sigma_v \rangle \rightarrow_v r_v \Longrightarrow r_v \neq \mathsf{F}$$

Fig. 9. The correctness definitions for a Boogie procedure p (top) and Viper method m (bottom). We ignore the restriction on well-typed states here, but include the restriction in the Isabelle formalisation.

- Global variables H and M to model the heap and permission mask, respectively. $\mathsf{H[x,f]}$ stores the heap value for heap location $\mathsf{x.f}$ and $\mathsf{M[x,f]}$ stores the permission value for $\mathsf{x.f}$. The types of both variables are represented via Boogie's *impredicatively-polymorphic maps* [30], which we explain below.

The correctness of a Boogie procedure guarantees no failing executions of the procedure's body for *any* interpretation of the uninterpreted types and functions (1) that is well-formed (e.g. the function interpretation respects the declared function signatures), and (2) for which all the Boogie axioms in the Boogie program are satisfied in the initial Boogie state. The formal correctness definition for a Boogie procedure p reflects this directly (a simplified version is shown at the top of Fig. 9). $\mathcal{T}$ and $\mathcal{F}$ are the interpretations of uninterpreted types and functions, respectively. G denotes the global declarations in the Boogie program. $\text{init}_b(p)$ is the initial Boogie program point in the procedure p. $\text{initCtxt}_b^G(p, \mathcal{T}, \mathcal{F})$ constructs a Boogie context from the provided parameters. Thus, to use the correctness of a Boogie procedure, we must choose a type and function interpretation that satisfy the required conditions. The main challenge here is formally expressing interpretations that deal with polymorphic Boogie maps, as we discuss next.

Polymorphic maps. The heap and permission maps are represented (via the Viper-to-Boogie translation) using Boogie's polymorphic maps; this choice is not unusual (e.g. the Dafny-to-Boogie implementation also currently uses polymorphic maps with similar polymorphic map types as the ones used by the Viper-to-Boogie implementation). The Boogie maps used to model Viper heaps have the polymorphic map type `<T>[bref, bfield T]T`: a total map storing, for *any* type T, values of type T given (as key) a reference and field with type argument T.

To our knowledge, there exists no formal model for Boogie's polymorphic maps. Providing a general model is challenging: in particular, Boogie's polymorphic maps are *impredicative*: a map m of type `<T>[T]T'` permits *any* value as a key, including the map m itself! Instead of providing a formal model for polymorphic maps in general, we provide one tailored to those that the Viper-to-Boogie implementation uses. To aid the incorporation of our model, we adjust the implementation to represent a polymorphic map via an uninterpreted type (e.g. `HType` for the heap), polymorphic functions for reading from and updating a polymorphic map (e.g. `read` and `upd`), and two axioms that express the relationship between the two functions. The only change in the translation itself is to simply rewrite heap and mask lookups and updates into calls to these functions; everything else remains identical. Then, we provide interpretations of the types and functions, and automatically prove that the axioms hold for these interpretations for any state that maps constants to their defined values; the same approach could be used for e.g. the Dafny-to-Boogie translation.

What remains for our simulation proofs is to provide interpretations for these new components (e.g. `HType`, `read`, and `upd` for the heap) such that the axioms are fulfilled. The challenge here
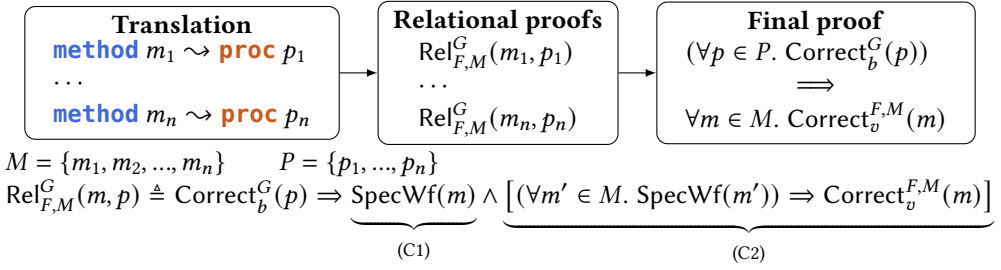
$$\boxed{\begin{array}{c} \textbf{Translation} \\ \mathtt{method}\ m_1 \rightsquigarrow \mathtt{proc}\ p_1 \\ \ldots \\ \mathtt{method}\ m_n \rightsquigarrow \mathtt{proc}\ p_n \end{array}} \longrightarrow \boxed{\begin{array}{c} \textbf{Relational proofs} \\ \mathrm{Rel}^G_{F,M}(m_1, p_1) \\ \ldots \\ \mathrm{Rel}^G_{F,M}(m_n, p_n) \end{array}} \longrightarrow \boxed{\begin{array}{c} \textbf{Final proof} \\ (\forall p \in P.\ \mathrm{Correct}^G_b(p)) \\ \Longrightarrow \\ \forall m \in M.\ \mathrm{Correct}^{F,M}_v(m) \end{array}}$$

$$M = \{m_1, m_2, ..., m_n\} \qquad P = \{p_1, ..., p_n\}$$

$$\mathrm{Rel}^G_{F,M}(m,p) \triangleq \mathrm{Correct}^G_b(p) \Rightarrow \underbrace{\mathrm{SpecWf}(m)}_{(C1)} \wedge \underbrace{\left[(\forall m' \in M.\ \mathrm{SpecWf}(m')) \Rightarrow \mathrm{Correct}^{F,M}_v(m)\right]}_{(C2)}$$

Fig. 10. Proof strategy for the Viper-to-Boogie translation. First, a proof is generated relating each Viper method with the corresponding Boogie procedure. Second, the final proof is deduced. F denotes the Viper fields, G denotes the global constants, variables, Boogie axioms, and functions emitted by the translation.

is avoiding circularities: e.g. if the field provided to `read` has type parameter `HType`, then the instantiation of `read` must itself return a heap; to construct an initial heap, we already need a heap of the same type. To break this circularity, we instantiate `HType` as a *partial* mapping from reference and fields to values, and allow the empty map to be of type `HType`, which provides us with a concrete heap. `read` is defined to return a default value for reference and field pairs that are not in the domain of the partial map; for heaps the default value is the empty map. This is sufficient to prove the axioms, since in practice the axioms only require `read` returning specific values when those values were previously inserted by `upd`.

4.5 Generating A Proof of the Final Theorem

We will now discuss, given a Viper program and its Boogie translation, how forward simulation proofs can be used to generate a proof of the final theorem justifying the soundness of the translation: The correctness of the Boogie program (i.e. the correctness of all contained Boogie procedures) implies the correctness of the Viper program (i.e. the correctness of all contained Viper methods).

We decompose the proof of the final theorem into smaller parts. At a high level, the Viper-to-Boogie translation works as follows. Let F and M be the set of Viper fields and methods in the Viper program, respectively. The Viper-to-Boogie translation (1) emits global Boogie declarations G (see Sec. 4.4) and (2) generates a separate Boogie procedure $p(m)$ for every Viper method m in M. The intended relation between m and $p(m)$ is given by $\mathrm{Rel}^G_{F,M}(m, p(m))$ in Fig. 10, which states that the correctness of $p(m)$ w.r.t. G guarantees two things: (C1) the well-formedness of m's specification, and (C2) the correctness of m w.r.t. F and M *if* the specifications of all methods in the Viper program are well-formed. The reason that the correctness of m is not implied *directly* is due to the optimised translation of method calls (as explained in Sec. 4.2).

Fig. 10 shows how we generate the proof of the desired theorem in two steps. First, for each Viper method m and its translated Boogie procedure $p(m)$, we generate a proof for $\mathrm{Rel}^G_{F,M}(m, p(m))$, explained next. Second, we obtain the desired theorem directly from these per-method relational proofs, since the correctness of all Boogie procedures implies that all Viper method specifications are well-formed using (C1), which implies that each Viper method is correct using (C2).

Next, we turn the focus to our strategy for proving $\mathrm{Rel}^G_{F,M}(m, p(m))$. For the sake of presentation, we focus on the proof of (C2) (correctness of m), and omit the proof of (C1) (well-formedness of m's specification). Intuitively, to prove that m is correct, we have to show that for any state that satisfies m's precondition, executing m's body in this state results in a state that satisfies m's postcondition. The correctness definition for a Viper method (shown at the bottom of Fig. 9) expresses this by requiring that any execution starting in a state σ_v with no permissions that inhales the precondition,

then executes the body, and finally exhales the postcondition, cannot fail. As planned, we obtain this result via a forward simulation proof between the executed Viper statement and $p(m)$'s procedure body using our presented methodology. Formally, we show:

$$\exists R', \gamma'. \text{ stmSim}_{\Gamma_v^0, \Gamma_b^0}(R_0, R', s_v^0, \text{init}_b(p(m)), \gamma')$$

$$\text{where } s_v^0 \triangleq \textbf{inhale } \text{pre}(m); \text{body}(m); \textbf{exhale } \text{post}(m)$$

In the statement above, $\Gamma_v^0 \triangleq \text{initCtxt}_v^{M,F}(m)$ is the initial Viper context. Γ_b^0 is a Boogie context that is defined in terms of our chosen type and function interpretation (see Sec. 4.4). R_0 is an instantiation of the state relation shown in Sec. 4.1. $\text{init}_b(p(m))$ is the initial Boogie program point in $p(m)$. The output state relation and output Boogie program point are irrelevant, since we care only about the simulation of failing Viper executions here. To complete the proof, we choose an initial Boogie state σ_b such that $R_0(\sigma_v, \sigma_b)$. As a result, if a Viper execution E_v of statement s_v^0 in σ_v *fails*, the forward simulation provides us with a failing Boogie execution E_b of $p(m)$. Using the correctness of $p(m)$, we conclude that E_b cannot fail, and thus conclude that E_v cannot fail, which concludes the proof of $\text{Rel}_{F,M}^G(m, p(m))$.

5 IMPLEMENTATION AND EVALUATION

We instrumented the existing Viper verifier implementation to automatically produce an Isabelle proof justifying the soundness of its translation to Boogie, and evaluated this validation on a diverse set of Viper benchmarks.

Implementation. Even though Viper passes the generated Boogie program to Boogie as a text file, our soundness proof directly connects the input Viper AST to the internal AST representation of the Boogie verifier. Therefore, we do not have to trust the Boogie parser.

We make the following four adjustments to the Viper verifier implementation. First, we desugar the uses of polymorphic maps as described in Sec. 4.4, since there is no formal model for polymorphic maps. Second, we adjust the implementation to not emit Boogie declarations or commands that are used only for features outside of our subset (the implementation always emits those without checking whether the corresponding features are actually used). Third, we switch off simple syntactic transformations that the Viper verifier applies to the produced Boogie program (e.g. constant folding, elimination of if-statements with no branches), since we do not support them yet; justifying those transformations should be straightforward and is orthogonal to our work. Fourth, we introduce a **havoc** statement in the Boogie program at the point when a scoped Viper variable is introduced, which faithfully models the semantics of such a variable. The original Viper implementation instead just introduces a fresh Boogie variable at the beginning of the Boogie procedure. Proving the equivalence of both translations is straightforward.

Evaluation. Our evaluation answers the questions: **(RQ1)** Does our implementation generate proofs that Isabelle can check successfully for a diverse set of examples? **(RQ2)** Does Isabelle check the generated proofs in reasonable time (e.g. feasible as part of continuous integration)?

To obtain a diverse set of representative examples, we considered the Viper test suite as well as the test suites of three tools that produce Viper code: Gobra [47] (for Go), VerCors [4] (for Java), and MPP [15] (a tool performing a modular product transformation on Viper programs). To eliminate trivial translations, we focused on Viper programs that use the heap, as indicated by the occurrence of at least one accessibility predicate. Out of those, we included all Viper programs that fall into our supported Viper subset. We followed different strategies to systematically obtain additional examples from the different test suites. For Viper and MPP, we additionally included all files that have an *old*-expression (by manually removing the corresponding assertion, i.e. verifying weaker

Table 1. Overview of benchmarks and results. For each test suite, we report the number of Viper files, the total number of Viper methods contained in those files, as well as the *mean* number of non-empty lines of code for the Viper files, Boogie files, and produced Isabelle proofs. We measured the mean and median time it took to check the Isabelle proofs in seconds.

Test suite	Files no.	Methods no.	Viper Mean [LoC]	Boogie Mean [LoC]	Isabelle Mean [LoC]	Proof Check Mean [s]	Median [s]
Viper	34	105	33	298	1719	33.8	23.8
Gobra	17	65	60	287	1937	32.7	25.3
VerCors	18	116	63	332	2930	43.1	40.9
MPP	3	13	206	1060	5164	109.0	46.2
Overall	**72**	**299**	**54**	**335**	**2217**	**39.0**	**32.9**

Table 2. Detailed results of our evaluation for a selection of files showing the number of methods, the nonempty lines of code for the Viper program, Boogie program, and produced Isabelle proof, and the time it took to check the proof in seconds.

Test suite	File	Methods no.	Viper Total [LoC]	Boogie Total [LoC]	Isabelle Total [LoC]	Proof Check Total [s]
Viper	testHistoryProcesses	13	205	1711	7035	126.3
Gobra	defer-simple-02	9	211	853	4717	60.6
VerCors	inv-test-fail2	5	92	514	2596	56.5
MPP	banerjee	8	414	2014	9545	242.4
MPP	darvas	2	91	582	2800	38.4
MPP	kusters	3	112	583	3146	46.2

postconditions) or a *new* statement (by manually desugaring the allocation primitive into our subset). Moreover, we made sure that each argument to a method call is a variable (e.g. we rewrote `m(i+1)` to `var t := i+1; m(t)`), since we currently support only variables as arguments. For Gobra and VerCors, we removed boilerplate code that is emitted for each file and then followed the same process as for Viper and MPP. Moreover, we additionally included files generated by Gobra that had at most two occurrences of features outside of our subset if those could be desugared into our subset (e.g. by eliminating a function by inlining its body).

As summarised in Tab. 1, we collected a total of 72 Viper files (containing 299 methods), with a mean of 54 and maximum of 414 non-empty lines of code. We ran our implementation on all 72 Viper files to generate the Boogie translations and the Isabelle proofs, and measured the time it took for Isabelle to check the generated proofs (the mean of five repetitions). The measurements were run on a ThinkPad X1 Yoga Gen 5 on Ubuntu 20.04 with 16 GB RAM and i7-10510U @ 1.8 GHz (scaled up to 4.9 GHz using Turbo Boost). The generated Boogie translations are on average 6.2x larger (335 non-empty LoC on average), illustrating the semantic gap between Viper and Boogie.

Isabelle successfully checked the generated proofs for all Viper files, including the Viper programs automatically generated by other tools. This shows that our approach is effective for practical verifiers and answers RQ1 positively. The resulting Isabelle proofs have on average over 2000 lines and are checked in less than a minute.

Tab. 2 shows the results for a selection of examples (the detailed results for each test suite are shown in App. D of the TR [38]): All three examples from MPP, as well as the largest example (in

terms of lines of Viper code) from each of the other test suites. The three MPP examples are drawn from different research papers and show that our tool can certify challenging programs.

For this selection, the times to check the proofs range from 38 seconds to 4 minutes. No file in any of the 72 files takes longer than 4 minutes to check. These times are acceptable, since we expect the validation to be performed occasionally (in particular, before the verified program is released or as part of continuous integration), but not on every run of the verifier. Thus, we answer RQ2 positively for the considered 72 files. To obtain additional representative files from the test suites, we would need to extend the supported Viper subset. Moreover, most of our proofs are not yet optimised to make proof checking faster. For example, field and variable accesses currently result in an overhead in the proof that is proportional to the number of fields and active variables, respectively. This could be improved by constructing and updating lookup tables efficiently.

6 RELATED WORK

Various works prove the soundness of front-end translations *once and for all*. For instance, Lehner and Müller [27] prove a simplified translation from Java Bytecode to Boogie, and Vogels et al. [44] target a translation from a toy object-oriented programming language to Boogie. Both proofs are done on paper and do not consider an actual implementation of the translation. Backes et al. [3] prove a translation sound from the Dminor data processing language to the Bemol IVL in Coq. They do not provide a proof connecting the formalised translation to their F# implementation. Herms [21] proves a translation from C to the WhyCert IVL (inspired by the Why3 IVL) sound in Coq, which they then turn into an executable tool via Coq's extraction to OCaml. The resulting tool has similarities to the Jessie Frama-C implementation [33], which translates C programs to Why3; Herms [21] discusses mismatches between their mechanisation and the Jessie implementation. In contrast, our certification applies to existing front-end implementations, which are typically implemented in efficient mainstream programming languages, use diverse libraries, and include subtle optimisations omitted from idealised implementations. Smans et al. [41] prove soundness of a verification condition generator for a language with implicit dynamic frames (IDF) assertions once and for all on paper without using an IVL. They also implement a prototype, but do not formally connect the proof to the implementation. We also applied our methodology to a verifier based on IDF, but validate an actual implementation.

Many verifiers perform a series of program transformations, e.g. by translating programs to a lower-level IVL or internally without changing the language (e.g. monomorphisation). Our approach can in principle be applied to both kinds of transformations, but is tailored towards the former, where the semantic gap is large, non-local checks arise, and diverse translations are used. Existing work for the validation of internal transformations does not provide solutions for these challenges. For instance, our prior work [39] validates the verification condition (VC) generation implementation of Boogie programs (also via a proof-producing instrumentation), which includes internal Boogie-to-Boogie transformations. In these transformations, the semantic gap is small (the source and target constructs are largely the same), and thus the decomposition into smaller problems is immediate, while in this paper the decomposition is a challenge. Moreover, our prior work need not deal with non-local checks and diverse translations. Our prior work uses different kinds of simulations; it would be interesting future work to apply our methodology to these. Besides internal transformations, our prior work connects the VC and a Boogie program; this paper considers only program-to-program transformations. Our prior work can in principle be combined with this paper to enable end-to-end soundness guarantees for Viper, but requires extending the Boogie verifier validation to more internal Boogie transformations and to a larger Boogie subset.

Validation has also been used to obtain formal guarantees for implementations of other verifiers, but none of the existing works target front-end translations and the challenges they entail. Lin

et al. [31] and Wils and Jacobs [45] validate verifiers obtained via the K framework and VeriFast, respectively. These verifiers use symbolic execution, which requires a fundamentally different validation approach. Garchery [19] validates certain logical transformations in Why3. Cohen and Johnson-Freyd [9] also prove such logical transformations, but do so once and for all in Coq to demonstrate their Why3 mechanisation. Neither of the two consider the actual VC generation.

Multiple works also embed programs in an interactive theorem prover (ITP) and then automate forward simulation proofs. Rizkallah et al. [40] define a refinement calculus for the Cogent compiler to automatically prove a forward simulation in Isabelle for a Cogent expression and its C translation. Their calculus includes syntax-directed rules for deriving simulation judgements, but these rules do not provide the abstraction we needed to handle diverse translations. The compiler was developed with validation in mind, which simplifies, for instance, the treatment of optimisations. In contrast, our goal was to validate existing verifier implementations with all their intricacies. The verification of the seL4 kernel includes two large forward simulation proofs in Isabelle, for which proof automation was developed [8, 24, 46]. This automation reduces the manual proof overhead, but still requires user interaction. In contrast, our validation proofs are generated and checked completely automatically. Like us, they prove rules to decompose the simulation for composite statements syntactically but, contrary to us, do not decompose statements semantically into smaller simulations. They turn certain simulation judgements into Hoare triples for which they have separate automation.

Formal translation validation approaches for compilers express a per-run validator in an ITP [20, 42, 43], prove it correct once and for all, and then extract executable code (the extraction must be trusted). For many of these validators, the source and target languages are similar. It would be interesting to test the feasibility of such approaches for front-end translations, where the semantic gap between the languages is large. Another difference is that front-end translations incorporate reasoning steps, such as assumptions and proof obligations prescribed by a program logic. This encoding is achieved via components not present in executable languages such as assume statements, havoc statements, and axiomatisations. Moreover, front-end translations emit code that checks nontrivial properties that are then relied upon in other parts of the encoding.

Zimmerman et al. [48] define a formal Viper semantics for a Viper subset in order to prove formal results for the gradual verifier Gradual C0 that uses Viper. However, in contrast to ours, their formalisation is not mechanised. Boogie developers have added an option to monomorphise polymorphic maps in Boogie programs via non-polymorphic maps [11]. This option provides an alternative to ours for desugaring polymorphic maps, which, in the case of Viper, circumvents the circularity challenge discussed in Sec. 4.4, since Viper does not permit storing heaps in fields. However, in general, front-ends may permit storing heaps in fields.

7 CONCLUSION

We presented a methodology for the validation of the front-end translations implemented in practical automated program verifiers. We demonstrated that it handles the complexity and intricacies of the Viper-to-Boogie translation as implemented in the Viper tool. To the best of our knowledge, this is the first formal soundness guarantee for a practical front-end translation. Together with existing work on back-end (and SMT) validation, our work provides a path towards trustworthy automated verifiers. Two fundamental requirements of our approach are the existence of a formal semantics for the input language and IVL, and the ability to instrument the verifier implementation. As future work, we plan to extend the supported Viper subset and to apply our methodology to verifiers that target Viper as an IVL and that verify, for instance, concurrent or object-oriented programs.

ACKNOWLEDGMENTS

DATA AVAILABILITY STATEMENT

Our publicly-available artifact [37] contains:

(1) an Isabelle formalisation for the technical results in Sec. 2, Sec. 3, and Sec. 4.
(2) our proof-producing Viper-to-Boogie implementation, which generates, on every run of the verifier, an Isabelle proof showing that the correctness of the input Viper program is implied by the correctness of the corresponding Boogie translation.
(3) the examples used in the evaluation, and scripts for the benchmark selection and evaluation results (Sec. 5).

REFERENCES

[1] Andrew W. Appel and Sandrine Blazy. 2007. Separation Logic for Small-Step cminor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4732)*, Klaus Schneider and Jens Brandt (Eds.). Springer, 5–21. https://doi.org/10.1007/978-3-540-74591-4_3

[2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147, 30 pages. https://doi.org/10.1145/3360573

[3] Michael Backes, Catalin Hritcu, and Thorsten Tarrach. 2011. Automatically Verifying Typing Constraints for a Data Processing Language. In *Certified Programs and Proofs (CPP)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). https://doi.org/10.1007/978-3-642-25379-9_22

[4] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods (IFM)*, Nadia Polikarpova and Steve Schneider (Eds.). https://doi.org/10.1007/978-3-319-66845-1_7

[5] Sascha Böhme and Tjark Weber. 2010. Fast LCF-Style Proof Reconstruction for Z3. In *Interactive Theorem Proving (ITP)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). https://doi.org/10.1007/978-3-642-14052-5_14

[6] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis (SAS)*, Radhia Cousot (Ed.). 55–72. https://doi.org/10.1007/3-540-44898-5_4

[7] Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamaric, and Michael Emmi. 2016. SMACK software verification toolchain. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 589–592. https://doi.org/10.1145/2889160.2889163

[8] David A. Cock, Gerwin Klein, and Thomas Sewell. 2008. Secure Microkernels, State Monads and Scalable Refinement. In *Theorem Proving in Higher Order Logics (TPHOLS)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). https://doi.org/10.1007/978-3-540-71067-7_16

[9] Joshua M. Cohen and Philip Johnson-Freyd. 2024. A Formalization of Core Why3 in Coq. *Proc. ACM Program. Lang.* 8, POPL, Article 60 (jan 2024), 30 pages. https://doi.org/10.1145/3632902

[10] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *International Conference on Formal Engineering Methods (ICFEM)*, Adrián Riesco and Min Zhang (Eds.), Vol. 13478. 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

[11] Boogie developers. 2022. Monomorphization of polymorphic maps and binders. https://github.com/boogie-org/boogie/pull/669 Accessed March 19, 2024.

[12] Viper developers. 2024. Viper-to-Boogie implementation. https://github.com/viperproject/carbon Accessed April 4, 2024.

[13] Jenna DiVincenzo, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2022. Gradual C0: Symbolic Execution for Efficient Gradual Verification. *CoRR* abs/2210.02428 (2022). https://doi.org/10.48550/ARXIV.2210.02428 arXiv:2210.02428

[14] Marco Eilers and Peter Müller. 2018. Nagini: A Static Verifier for Python. In *Computer Aided Verification (CAV)*, Hana Chockler and Georg Weissenbacher (Eds.). https://doi.org/10.1007/978-3-319-96145-3_33

[15] Marco Eilers, Peter Müller, and Samuel Hitz. 2018. Modular Product Programs. In *European Symposium on Programming (ESOP)*, Amal Ahmed (Ed.). https://doi.org/10.1007/978-3-319-89884-1_18

[16] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification (CAV)*, Rupak Majumdar and Viktor Kuncak (Eds.). https://doi.org/10.1007/978-3-319-63390-9_7

[17] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *European Symposium on Programming (ESOP)*, Matthias Felleisen and Philippa Gardner (Eds.). https://doi.org/10.1007/978-3-642-37036-6_8

[18] Mathias Fleury and Hans-Jörg Schurr. 2019. Reconstructing veriT Proofs in Isabelle/HOL. In *Workshop on Proof eXchange for Theorem Proving (PxTP)*, Giselle Reis and Haniel Barbosa (Eds.). https://doi.org/10.4204/EPTCS.301.6

[19] Quentin Garchery. 2021. A Framework for Proof-carrying Logical Transformations. In *Workshop on Proof eXchange for Theorem Proving (PxTP)*, Chantal Keller and Mathias Fleury (Eds.). https://doi.org/10.4204/EPTCS.336.2

[20] Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. 2023. Formally Verifying Optimizations with Block Simulations. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 224 (oct 2023), 30 pages. https://doi.org/10.1145/3622799

[21] Paolo Herms. 2013. *Certification of a Tool Chain for Deductive Program Verification. (Certification d'une chaine de vérification déductive de programmes)*. Ph. D. Dissertation. University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00789543

[22] Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *Formal Methods (FM)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). https://doi.org/10.1007/11813040_19

[23] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

[24] Gerwin Klein, Thomas Sewell, and Simon Winwood. 2010. Refinement in the Formal Verification of the seL4 Microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, David S. Hardin (Ed.). Springer, 323–339. https://doi.org/10.1007/978-1-4419-1539-9_11

[25] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 712–717. https://doi.org/10.1007/978-3-642-31424-7_54

[26] Akash Lal and Shaz Qadeer. 2014. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 202–212. https://doi.org/10.1145/2635868.2635894

[27] Hermann Lehner and Peter Müller. 2007. Formal Translation of Bytecode into BoogiePL. *Electronic Notes in Theoretical Computer Science* 190, 1 (2007), 35–50. https://doi.org/10.1016/j.entcs.2007.02.059 Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2007).

[28] K. Rustan M. Leino. 2008. This is Boogie 2. (2008). Available from http://research.microsoft.com/en-us/people/leino/papers/krml178.pdf.

[29] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, Edmund M. Clarke and Andrei Voronkov (Eds.). https://doi.org/10.1007/978-3-642-17511-4_20

[30] K. Rustan M. Leino and Philipp Rümmer. 2010. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Javier Esparza and Rupak Majumdar (Eds.). https://doi.org/10.1007/978-3-642-12002-2_26

[31] Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Rosu. 2023. Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 56–84. https://doi.org/10.1145/3586029

[32] Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233. https://doi.org/10.1006/inco.1995.1134

[33] Claude Marché and Yannick Moy. 2018. The Jessie plugin for Deductive Verification in Frama-C. http://krakatoa.lri.fr/jessie.pdf

[34] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). https://doi.org/10.1007/978-3-662-49122-5_2

[35] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

[36] Matthew J. Parkinson and Alexander J. Summers. 2012. The Relationship Between Separation Logic and Implicit Dynamic Frames. *Logical Methods in Computer Science* 8, 3:01 (2012), 1–54. https://doi.org/10.2168/LMCS-8(3:1)2012

[37] Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. 2024. *Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language – Artifact.* https://doi.org/10.5281/zenodo.10802176

[38] Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. 2024. Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language (extended version). https://doi.org/10.48550/ARXIV.2404.03614 arXiv:2404.03614 [cs.PL]

[39] Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In *Computer Aided Verification (CAV) (LNCS, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). 704–727. https://doi.org/10.1007/978-3-030-81688-9_33

[40] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby C. Murray, Gabriele Keller, and Gerwin Klein. 2016. A Framework for the Automatic Formal Verification of Refinement from Cogent to C.. In *Interactive Theorem Proving (ITP)*, Jasmin Christian Blanchette and Stephan Merz (Eds.). https://doi.org/10.1007/978-3-319-43144-4_20

[41] Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *Transactions on Programming Languages and Systems (TOPLAS)* 34, 1, Article 2 (May 2012), 58 pages. https://doi.org/10.1145/2160910.2160911

[42] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Principles of Programming Languages (POPL)*, George C. Necula and Philip Wadler (Eds.). https://doi.org/10.1145/1328438.1328444

[43] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified validation of lazy code motion. In *Programming Language Design and Implementation (PLDI)*, Michael Hind and Amer Diwan (Eds.). https://doi.org/10.1145/1542476.1542512

[44] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2009. A Machine Checked Soundness Proof for an Intermediate Verification Language. In *Theory and Practice of Computer Science, Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM) (Lecture Notes in Computer Science, Vol. 5404)*, Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia (Eds.). Springer, 570–581. https://doi.org/10.1007/978-3-540-95891-8_51

[45] Stefan Wils and Bart Jacobs. 2023. Certifying C program correctness with respect to CH2O with VeriFast. *CoRR* abs/2308.15567 (2023). https://doi.org/10.48550/ARXIV.2308.15567 arXiv:2308.15567

[46] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David A. Cock, and Michael Norrish. 2009. Mind the Gap. In *Theorem Proving in Higher Order Logics (TPHOLS)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). https://doi.org/10.1007/978-3-642-03359-9_34

[47] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification (CAV)*, Alexandra Silva and K. Rustan M. Leino (Eds.). https://doi.org/10.1007/978-3-030-81685-8_17

[48] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *Proc. ACM Program. Lang.* 8, POPL (2024), 2547–2576. https://doi.org/10.1145/3632927