Verification-Preserving Inlining in Automatic Separation Logic Verifiers

THIBAULT DARDINIER, ETH Zurich, Switzerland
GAURAV PARTHASARATHY, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland

Bounded verification has proved useful to detect bugs and to increase confidence in the correctness of a program. In contrast to unbounded verification, reasoning about calls via (bounded) inlining and about loops via (bounded) unrolling does not require method specifications and loop invariants and, therefore, reduces the annotation overhead to the bare minimum, namely specifications of the properties to be verified. For verifiers based on traditional program logics, verification is preserved by inlining (and unrolling): successful unbounded verification of a program w.r.t. *some* annotation implies successful verification of the inlined program. That is, any error detected in the inlined program reveals a true error in the original program. However, this essential property might not hold for *automatic separation logic* verifiers such as CAPER, GRASSHOPPER, REFINEDC, STEEL, VERIFAST, and verifiers based on VIPER. In this setting, inlining generally changes the resources owned by method executions, which may affect automatic proof search algorithms and introduce spurious errors.

In this paper, we present the first technique for *verification-preserving* inlining in automatic separation logic verifiers. We identify a semantic condition on programs and prove in ISABELLE/HOL that it ensures verification-preserving inlining for state-of-the-art automatic separation logic verifiers. We also prove a dual result: successful verification of the inlined program ensures that there are method and loop annotations that enable the verification of the original program for bounded executions. To check our semantic condition automatically, we present two approximations that can be checked syntactically and with a program verifier, respectively. We implement these checks in VIPER and demonstrate that they are effective for non-trivial examples from different verifiers.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**; **Automated reasoning**.

Additional Key Words and Phrases: Modular Verification, Bounded Verification, Inlining, Loop Unrolling

ACM Reference Format:
Thibault Dardinier, Gaurav Parthasarathy, and Peter Müller. 2023. Verification-Preserving Inlining in Automatic Separation Logic Verifiers. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 102 (April 2023), 30 pages. https://doi.org/10.1145/3586054

1 INTRODUCTION

Modular deductive program verification can reason about complex programs and properties, but is expensive. Even automatic modular verifiers require a substantial annotation overhead, including method pre- and postconditions, loop invariants, and often ghost code. Bounded verification is a powerful alternative that reduces this overhead significantly. By inlining method calls (i.e., replacing a call by the callee's body up to a finite call depth), bounded verification does not require method

102

Authors' addresses: Thibault Dardinier, thibault.dardinier@inf.ethz.ch, Department of Computer Science, ETH Zurich, Switzerland; Gaurav Parthasarathy, gaurav.parthasarathy@inf.ethz.ch, Department of Computer Science, ETH Zurich, Switzerland; Peter Müller, peter.mueller@inf.ethz.ch, Department of Computer Science, ETH Zurich, Switzerland.

2475-1421/2023/4-ART102
https://doi.org/10.1145/3586054

specifications. Similarly, it avoids the need for loop invariants by unrolling loops (i.e., replacing a loop by finitely many copies of its body).

While bounded verification generally does not prove correctness for all executions of a program, it effectively finds errors and increases the confidence that a program is correct. Consequently, bounded verification is commonly applied by model checkers [Clarke et al. 2004] and also used by deductive verifiers. For example, the Corral verifier [Lal et al. 2012], which powers Microsoft's Static Driver Verifier [Lal and Qadeer 2014], inlines method calls and unrolls loop iterations in a Boogie program [Leino 2008], before calling the deductive Boogie verifier. In the following, we subsume both method inlining and loop unrolling under the term *inlining*.

Inlining is also a useful stepping stone toward *modular verification*, where we use "modular verification" to refer to unbounded verification that verifies method calls (resp. loops) w.r.t. their annotated contracts (resp. annotated loop invariants). Detecting errors using inlining *before* adding method specifications and loop invariants can prevent developers from wasting time attempting to annotate and verify an incorrect program. Inlining is also useful *during* the process of adding annotations to validate partial annotations before the program contains sufficient annotations to enable modular verification. For instance, inlining lets developers validate method specifications before providing loop invariants or validate partial loop invariants.

To avoid unnecessary manual labor, it is crucial that inlining itself does not introduce false positives (spurious errors): a verification error in the inlined program should occur only if the error occurs also during the verification of the original program *for all method specifications and loop invariants*. Equivalently, if there exist annotations (possibly extending existing partial annotations) s.t. the original program verifies then the inlined program must also verify. If this holds, we say that inlining is *verification-preserving*. Note that even with verification-preserving inlining, bounded or modular verification may report false positives if the program logic on which the verifier is based is incomplete or if valid proof obligations generated by the verifier cannot be discharged (e.g., due to limitations of SMT solvers); however, those false positives are not caused by inlining and, thus, irrelevant here. Verification-preserving inlining ensures that errors detected by a verifier in the inlined program will cause the verifier to also reject the original program. This property increases the confidence that the original program is actually incorrect and spares developers the effort of trying to find (non-existing) annotations that make modular verification succeed.

In verifiers based on traditional program logics (like Corral), inlining is trivially verification-preserving. However, many *automatic* verifiers based on *separation logic* rely on proof search algorithms that may render inlining *non-preserving*. Separation logic [Reynolds 2002] (SL thereafter) uses resources, such as permissions to access heap locations. These resources are owned by method executions and transferred between executions upon call and return. As a result, inlining a method call potentially changes the resources available during the verification of the callee's method body, which may affect proof search algorithms that depend on the resources owned by a method execution, in particular, (1) the automatic instantiation and (2) the automatic selection of proof rules. Both automation techniques may cause inlining to be non-preserving, as we show in Sect. 2.

The usefulness of automatic SL verifiers (often based on SMT solvers) relies heavily on these automation techniques. Thus, these techniques, which may cause inlining to be non-preserving, are frequently used (in different forms) in diverse and independently-developed verifiers such as Caper [Dinsdale-Young et al. 2017] (a verifier for fine-grained concurrency), GRASShopper [Piskac et al. 2014] (a verifier for a decidable separation logic fragment), Steel [Fromherz et al. 2021] (a verifier based on F* [Swamy et al. 2016]), RefinedC [Sammler et al. 2021] (a verifier for C programs based on Iris [Jung et al. 2018]), VeriFast [Jacobs et al. 2011] (a verifier for C and Java programs), and verifiers built on top of the Viper infrastructure [Müller et al. 2016] such as Nagini [Eilers and

Müller 2018] (a verifier for Python programs), RSL-Viper [Summers and Müller 2018] (a verifier for C++ weak-memory programs), and VerCors [Blom et al. 2017] (a verifier for Java).

Our evaluation, performed on the test suites of GRASShopper, Nagini, RSL-Viper, and VeriFast, shows that, while most method calls (and loops) can be inlined in a verification-preserving manner, non-preserving inlining occurs in practice in all four verifiers. More precisely, a syntactic analysis of all files from their test suites shows that 1053 files (out of 1562, 67%) contain features that may result in non-preserving inlining. Further manual analysis of a sample of 72 files suggests that, for each verifier, between 10% and 67% of the sampled files contain methods (or loops) that actually result in non-preserving inlining for some caller context.

Approach. This paper presents the theoretical foundations for *verification-preserving* inlining in automatic separation logic verifiers. The core contribution is a novel semantic condition for programs that ensures that inlining is verification-preserving, even in the presence of the automation techniques mentioned above and described in more detail in the next section. A key virtue of this semantic condition is that it is compositional, whereas the definition of the verification-preserving property itself is not. Our semantic condition is inspired by the *safety monotonicity* and *framing* properties [Yang and O'Hearn 2002] of separation logics, but goes beyond those in three major ways: (1) We show that only a subset of statements must satisfy these properties for inlining to be verification-preserving. (2) Our semantic condition includes a novel monotonicity property on the final state of a statement execution. (3) Our semantic condition uses bounded relaxations of the properties that are weaker, but still sufficient to ensure verification-preserving inlining. All three improvements are crucial to support common use cases.

We have proved in Isabelle/HOL that our semantic condition is sufficient. Since it is difficult to check directly using automatic verifiers, we develop a *structural condition* that approximates the semantic condition and can be checked using SMT-based verification tools. We show its practicality by automating it in a tool that performs bounded verification of Viper programs via verification-preserving inlining. Errors reported by the resulting inlining feature are true errors. Our approach does not require pre- and postconditions and loop invariants, but checks partial annotations if present, which enables the use of inlining during the process of annotating a program. Our experiments show that the structural condition is sufficiently precise for most common use cases.

Contributions and outline. To the best of our knowledge, we present the first theoretical foundations of inlining in automatic SL verifiers. Our technical contributions are:

- We show why crucial automation techniques such as the automatic instantiation and the automatic selection of proof rules may cause inlining to be non-preserving (Sect. 2).
- We present a novel semantic condition for inlining in automatic SL verifiers. Programs that satisfy this condition are guaranteed to be inlined in a verification-preserving manner, without producing false positives. Our semantic condition takes partial annotations into account (Sect. 3).
- We formalize the semantic condition for a verification language that is parametrized by a separation algebra, to capture different state models and different flavors of SL, and prove that inlining is verification-preserving under our semantic condition in Isabelle/HOL (Sect. 4).
- We prove a dual result: inlining does not lead to false negatives other than errors that occur beyond the inlining bound [Dardinier et al. 2022b].
- We define a structural condition that approximates the semantic condition, but can be checked in SMT-based program verifiers (Sect. 5).
- We implement an inlining tool for Viper, which checks the structural condition and the correctness of the inlined program (Sect. 6).

- Our evaluation shows that (1) non-preserving inlining occurs in practice, (2) for many non-trivial examples, inlining is verification-preserving and our structural condition captures this, and (3) verification-preserving inlining is effective in practice (Sect. 6).

Our publicly-available artifact [Dardinier et al. 2023] contains Isabelle/HOL proofs of the technical results from Sect. 4, the tool that we implemented, and the examples used in the evaluation. Further details are available in our accompanying technical report (TR hereafter) [Dardinier et al. 2022b].

2 THE PROBLEM

In separation logic, resources (such as a permission to access a heap location) are owned by method executions, and transferred between executions upon call and return. Thus, inlining a method call potentially changes the resources that the callee owns, which may affect proof search algorithms that depend on the resources owned by the method execution. In this section, we show that this is the case for crucial automation techniques such as the automatic instantiation (Sect. 2.1) and the automatic selection (Sect. 2.2) of proof rules, by showing that both may cause inlining to be non-preserving in several automatic SL verifiers.

2.1 Automatic Instantiation of Proof Rules

Applying proof rules, for instance, instantiating quantifiers, often requires choosing a resource that is currently owned by the method. To handle a large, possibly unbounded search space, automatic SL verifiers employ heuristics for this purpose. These heuristics may behave differently when more resources are available and, thus, may make inlining non-preserving.

Such heuristics are often necessary when automatic SL verifiers support *imprecise assertions*. Imprecise assertions do not describe the extent of the heap precisely (e.g., because multiple disjoint heap fragments satisfy the same assertion): Caper, GRASShopper, RefinedC, Steel, and VeriFast support restricted forms of existentially-quantified assertions, VeriFast and Viper support fractional ownership of resources [Boyland 2003; Dardinier et al. 2022a] with existentially-quantified fractions, and RefinedC, Steel, and Viper [Dardinier et al. 2022c; Schwerhoff and Summers 2015] support magic wands. To prove the validity of such assertions, the majority of these verifiers use proof search algorithms (e.g., to choose a satisfying heap fragment among several suitable ones).

The effect of imprecise assertions on proof heuristics is relevant for inlining even though assertions are mainly needed for modular verification: First, as explained in Sect. 1, inlining is useful during the process of adding annotations to a program and, thus, must handle programs with partial annotations. Second, even bounded verification relies on method specifications for certain calls (e.g., to library methods and foreign functions). Both cases may involve imprecise assertions.

Fig. 1 illustrates the problem on an example inspired by VeriFast. It uses library methods `alloc` to create a data structure, and `crLock` and `acquire` to create and acquire a lock. The predicate `P` indicates that the lock is initialized; `Q` is the lock invariant (a real lock library would quantify over the lock invariant `Q`, but this aspect is irrelevant here). Inlining transforms method `m` into method `m_inl`, in which the call to `n` has been replaced by its body and where we assume that the user has not given any (partial) annotation to `n` (yet). Note that the calls to the library methods are not inlined because those methods are annotated and may be implemented natively.

`crLock`'s precondition contains an existentially-quantified predicate instance `Q(?x)` and is, thus, imprecise. Therefore, a proof search algorithm needs to decide how to instantiate the bound variable `x`. During modular verification, this choice is determined by `n`'s precondition. For instance, the precondition `Q(a)` will cause the proof search to instantiate `x` with `a` since `Q(a)` is the only matching resource held by `n`. With this precondition (and a trivial postcondition), the modular proof succeeds.

```
method alloc():(l:Ref)
  requires true
  ensures Q(l)

method m(r:Ref)
{
  { true }
  a := alloc()
  { Q(a) }
  b := alloc()
  { Q(a) * Q(b) }
  n(a)
  { ... * Q(b) }
}
```

```
method crLock():(l:Ref)
  requires Q(?x)
  ensures P(l,x)

method n(a:Ref)
  // requires Q(a)
  // ensures ...
{
  { Q(a) }
  l := crLock()
  { P(l,a) }
  acquire(l, a)
  { ... }
}
```

```
method acquire(l, a:Ref)
  requires P(l,a)
  ensures ...

method m_inl() {
  { true }
  a := alloc()
  { Q(a) }
  b := alloc()
  { Q(a) * Q(b) }
  l := crLock()
  { Q(a) * P(l,b) }
  acquire(l, a) // fails
}
```

Fig. 1. Example inspired by VeriFast showing that inlining in automatic SL verifiers is potentially non-preserving in the presence of imprecise assertions. The methods alloc, crLock, and acquire are part of a library and are specified via pre- and postconditions. Methods m and n are the client code. The commented-out specification of n illustrates one possible annotation with which m and n verify modularly. m_inl is the method m where the call to n is inlined. P and Q denote abstract predicates. Q(?x) denotes a predicate instance with an existentially-quantified parameter x. Proof outlines reflecting the verifier's automatically constructed proofs are shown in blue (where the proof outlines for m and n reflect modular proofs using n's commented-out specification). The verifier fails to verify the call to acquire in m_inl.

However, the proof search heuristic fails for the inlined program on the right. Here, the method m_inl owns Q(a) and Q(b) before the call to crLock. VeriFast's heuristic instantiates x with b. As a result, the call to acquire fails, since the method owns P(l,b) instead of P(l,a).

The fact that the program can be verified modularly, but fails to verify after inlining, shows that VeriFast's proof search heuristics make inlining non-preserving; GRASShopper's, RefinedC's, Viper's, and Steel's proof search heuristics for instantiating existential quantifiers also can lead to non-preserving inlining for similar reasons. In all these cases, non-preserving inlining is caused by heuristics for proof automation. The inlined program is correct and could be verified manually by instantiating the quantifier with a.

2.2 Automatic Selection of Proof Rules

Many advanced separation logics support proof rules that manipulate resources in intricate ways, e.g., to split and combine resources, to exchange resources, to put them under modalities, etc. Most of these proof rules can be applied at many points in the program and proof. To avoid exploring every possible combination, automatic SL verifiers use heuristics to decide when and how to apply the proof rules. Some of these heuristics are based on the resources currently owned by a method and may, thus, be affected by inlining, potentially making inlining non-preserving. For instance, Caper inspects the currently-owned resources to determine whether or not to create a shared memory region. RSL-Viper inspects the resources held by the method execution to determine the resources obtained from an atomic read operation. Both heuristics may make inlining non-preserving.

Fig. 2 illustrates a simplified version of a heuristic used by RSL-Viper, which makes inlining non-preserving. The commented-out specifications for read1 and read2 just serve to illustrate successful modular verification and are not (partial) annotations provided by a user. RSL-Viper automates the RSL logic [Vafeiadis and Narayan 2013], which associates an invariant—here represented by the predicate instance P(l)—with each atomic memory location l. The logic provides two proof

```
method r(l:Ref):(a,b:Int)
  requires P(l)
  ensures A(b)
{
  { P(l) }
  a := read1(l)
  { P(l) }
  b := read2(l)
  { A(b) }
}
```

```
method read1(l:Ref):(v:Int)
  // requires true
  // ensures true
{ v := [l]_acq
  v := v+1 }

method read2(l:Ref):(v:Int)
  // requires P(l)
  // ensures A(v)
{ v := [l]_acq }
```

```
v := nondetInt();
if(perm(P(l)) >= 1) {
  exhale P(l)
  inhale A(v)
}
```

Fig. 2. Example inspired by RSL-Viper showing that proof search algorithms may make inlining non-preserving. Method `r` performs two atomic read-acquire operations on the atomic memory location `l` and returns both values. Its specification summarizes the behavior of the code running before and after `r`. The precondition `P(l)` provides the invariant associated with location `l`. The postcondition `A(b)` indicates that the subsequent code requires the assertion that depends on the *second* value that has been read. The code on the far right shows the proof strategy for $v := [l]_{acq}$, expressed in the Viper intermediate language; it greedily exchanges the invariant for location `l` by an assertion for the read value. The program can be verified modularly by RSL-Viper using the commented-out specifications for `read1` and `read2`. The proof outline for `r` (shown in blue) reflects the corresponding proof by RSL-Viper for `r`. However, RSL-Viper's strategy makes inlining non-preserving: in the inlined program, it applies the exchange to the first read operation, whereas successful verification needs to apply it to the second.

rules for atomic reads (the complete RSL logic is more intricate). In one rule, the invariant is consumed (before the location is read) and instead an assertion that depends on the read value is obtained (after the read); this rule must be applied to verify `read2` modularly w.r.t. its (commented-out) annotation. In the other rule, no resource is consumed, and therefore the right to perform such an exchange via a (future) atomic read is retained by keeping the invariant (and thus, no assertion is obtained for the read value); this rule must be applied to verify `read1` modularly w.r.t. its (commented-out) annotation. RSL-Viper's proof strategy always attempts to apply the first proof rule before considering the second proof rule, that is, performs the exchange when possible. In our example, this greedy approach causes verification of the inlined program to fail because the first read consumes the invariant, such that no exchange can happen for the second read, and we do not obtain the assertion `A(b)` for the read value. However, the program can be verified modularly by *not* passing the invariant to method `read1` such that the heuristic is prevented from performing the exchange for the first read. In Fig. 2, the commented-out annotations for `read1` and `read2` serve to illustrate the annotation for this modular proof and the corresponding proof outline for `r` is shown in blue. The fact that the program can be verified modularly while verification of the inlined program fails shows that the proof search heuristic of RSL-Viper causes inlining to be non-preserving. The inlined program is correct and could be verified manually by applying the proof rule that exchanges the resources only for the second read operation, which demonstrates that the issue is, again, caused by heuristics for proof automation.

RSL-Viper is implemented by translating the input C++ program into the Viper intermediate language. The proof search algorithm is represented explicitly in the Viper program and a simplified version is shown on the right of Fig. 2. This snippet uses two dedicated statements to manipulate resources: **`inhale`** obtains resources and **`exhale`** releases resources. We sometimes use those operations in our examples, but no aspect of our work is specific to **`exhale`** and **`inhale`** operations. The same effects can be obtained, for instance, by calling a library method with a corresponding pre- or postcondition (as we do in Fig. 1). The code snippet also uses a *resource introspection expression*

perm. This expression yields the fractional ownership amount [Boyland 2003] held by the current method execution for a given resource and is used here to determine whether the resource `P(l)` is held by the current method execution.

3 SEMANTIC CONDITION: KEY IDEAS

In this section, we introduce the key ideas of the semantic condition that we define formally in Sect. 4 and under which we prove that inlining is verification-preserving. These sections focus on calls, but loops are handled in the Isabelle/HOL formalization. From a verification point of view, loops are analogous to recursive methods, where the loop invariant acts as both the pre- and postcondition of the method. Unrolling n loop iterations corresponds to inlining n recursive calls.

Verification-preserving inlining. Let M be a collection of methods and let s be an *initial statement* that may contain calls to methods in M (and no other calls). We call (s, M) a program and we do not mention the tuple explicitly whenever it is clear from the context. An *annotation* for M consists of a pre- and postcondition for each method in M. A program (s, M) verifies modularly w.r.t. an annotation $\mathcal{A}$ for M, if all methods in M verify modularly w.r.t. $\mathcal{A}$ and s verifies modularly w.r.t. to $\mathcal{A}$ (where method calls are verified using only their pre- and postconditions).

The *inlined version* of a program (s, M) with bound n is the statement s with all calls substituted by their bodies up to a call stack size of n (library calls may still be treated modularly). Calls that exceed the bound n are replaced by **assume false**, such that the code afterwards verifies trivially.

Inlining is *verification-preserving* for a program (s, M) with bound n if the following holds: If the program (s, M) verifies modularly w.r.t. *some* annotation, then the program inlined with bound n also verifies.[1] Consequently, if inlining is verification-preserving for a program then each error in the inlined program is a *true error*, i.e., corresponds to an error in the original program.

Semantic condition. The *semantic condition* (formalized in Def. 4.4, see Sect. 4) is a property of a program that guarantees (but is not equivalent to) verification-preserving inlining: if the semantic condition holds for a statement s, a collection of methods M, and an inlining bound n, then inlining the program (s, M) with bound n is verification-preserving. The semantic condition is parameterized by a *resource bound* (a set of states), which we explain in Sect. 3.4. Informally, the semantic condition holds iff:

(1) entire inlined method bodies satisfy *bounded framing*, and
(2) call-free statements between[2] method calls satisfy *bounded monotonicity*, which is defined as the conjunction of *bounded safety monotonicity* and *bounded output monotonicity*.

The rest of this section describes and illustrates the three key properties used in this definition: *framing* (Sect. 3.1), *safety monotonicity* (Sect. 3.2), and *output monotonicity* (Sect. 3.3).[3] Sect. 3.4 explains why it is sufficient to consider bounded relaxations of these properties. Finally, Sect. 3.5 shows how we deal with partially-annotated programs.

Verifier semantics. Our definition of verification-preserving inlining is based on the proof rules *as applied by a given SL verifier.* Thus, when we write that a program *verifies*, we mean that verification succeeds in a given verifier (using that verifier's proof search strategies). We refer to the proof rules as applied by a verifier as the *verifier semantics* of that tool. For example, verification of the inlined program from Fig. 1 in a verifier that does not apply VeriFast's proof search heuristic

[1] The definition of verification-preserving inlining is slightly different when the program already contains partial annotations as we discuss in Sect. 3.5 and Sect. 4.

[2] Statements before the first method call and after the last method call are also included.

[3] As we explain in this section, framing is stronger than both safety and output monotonicity; requiring these weaker properties for call-free statements is sufficient for inlining to be verification-preserving.

```
method client(x:Ref)
  requires P(x)
  ensures [0.5]P(x)
        * x.f ↦(0.5) _
{
  callee(x)
}
```

```
method callee(x:Ref)
  // requires [0.5]P(x)
  // ensures x.f ↦(0.5) _
{
  { [0.5]P(x) }
  open P(x)
  { x.f ↦(0.5) _ }
  var v := x.f
  { x.f ↦(0.5) _ }
}
```

```
method client_inl(x:Ref)
  requires P(x)
  ensures [0.5]P(x) // fails
        * x.f ↦(0.5) _
{
  { P(x) }
  open P(x)
  { x.f ↦ _ }
  var v := x.f
  { x.f ↦ _ }
}
```

Fig. 3. A simple example inspired by VeriFast showing that inlining (with bound 1) can be non-preserving if a method body (in this case `callee`'s body) is not framing. Predicate `P(x)` is a predicate instance with predicate body `x.f` $\mapsto$ _, which can be automatically split into two halves [Dardinier et al. 2022a]: `[0.5]P(x)` and `[0.5]P(x)`. `x.f` $\overset{f}{\mapsto}$ _ denotes the fractional points-to assertion, representing fractional ownership amount f for `x.f`. **open** `P(x)` is a ghost operation in VeriFast that exchanges all the ownership of a predicate instance held by the method execution for ownership of its body, which is needed to justify reading `x.f` on the next line. Both methods `client` and `callee` verify modularly with the commented-out specification. In the modular verification of `callee`, **open** `P(x)` removes half ownership of `P(x)`, whereas it removes the full ownership in the inlined version of `client` (shown on the right), which is why the postcondition in the inlined version of `client` does not verify.

could succeed and, thus, inlining could be verification-preserving. It is the automation embodied in the verifier semantics of the used verifier (here, VeriFast) that causes verification to fail.

Automatic SL verifiers track the resources held by a method execution. Thus, resources are a part of an SL verifier's state model, in addition to the program heap and store. For example, the state models of both Viper and VeriFast contain a mapping from heap locations (and predicate instances) to fractional permission amounts following the fractional permission extension of SL [Boyland 2003]. For heap locations, permissions are a fraction between 0 and 1 (a non-zero fraction permits reading, while the fraction 1 permits writing).

Program operations may observe and modify the held permissions. E.g., the resource introspection expression **perm**`(P(l))` on the right of Fig. 2 evaluates to the permission amount held in the verifier's state. The held permissions are modified, for instance, when an object is being allocated, via a method call that is treated modularly, or via an operation used to direct the verifier's proof search. We often model modifications of the held permissions via two dedicated statements: **inhale** `A` adds the resources specified by the assertion `A` to the state; **exhale** `A` removes these resources (following some heuristics when the assertion `A` is imprecise) or fails if they are not held in the current state.

3.1 Framing

Automatic SL verifiers verify method calls modularly by releasing (exhaling) the resources specified by the callee's precondition and then obtaining (inhaling) the resources specified by its postcondition. Resources held by the caller that are not exhaled are retained across the call, which is justified by separation logic's frame rule. The frame rule states that if the Hoare triple $\{P\}\ s\ \{Q\}$ holds then s also verifies in a larger state $P * R$, and the additional resources R remain unchanged, that is, the triple $\{P * R\}\ s\ \{Q * R\}$ holds (provided that s does not modify any variable in R).

However, the frame rule does not always apply to a verifier semantics, because of the heuristics and proof search algorithms used by automatic SL verifiers. As shown in Sect. 2, their behavior may depend on the resources held by the method execution. In particular, they may *not* preserve the additional resources R across an execution of s. As a result, even if a verifier is able to prove $\{P\}\ s\ \{Q\}$, it may *not* be able to prove $\{P * R\}\ s\ \{Q * R\}$, which may lead to non-preserving inlining.

To guarantee verification-preserving inlining, the semantic condition requires entire inlined method bodies to be *framing*.[4] Informally, a statement s is *framing* (in a verifier) iff for all P, Q, and R, if the verifier can prove that $\{P\}\ s\ \{Q\}$ holds then it can also prove that $\{P * R\}\ s\ \{Q * R\}$ holds. Framing is formally defined in Sect. 4 (Def. 4.3).

Fig. 3 shows a simple VeriFast example where inlining is non-preserving because `callee`'s body is not framing: VeriFast is able to prove $\{$`[0.5]P(x)`$\}$ **open** `P(x)` $\{$`x.f` $\overset{0.5}{\mapsto}$ _$\}$ (in the modular case in the middle), but not $\{$`[0.5]P(x)` $*$ `[0.5]P(x)`$\}$ **open** `P(x)` $\{$`x.f` $\overset{0.5}{\mapsto}$ _ $*$ `[0.5]P(x)`$\}$ (in the inlined case on the right). In other words, VeriFast does not frame the ownership of `P(x)` around **open** `P(x)`, since **open** `P(x)` consumes all the ownership of `P(x)` held by the method execution.

Both examples discussed in Sect. 2 contain method bodies that are not framing. In Fig. 1, method `n`'s body is not framing, because ownership of `Q(x)` (for some `x`) might be consumed by `crLock()`, and thus might not be framed around `crLock()` in the inlined program. Similarly, in Fig. 2, the bodies of methods `read1` and `read2` are not framing, because ownership of `P(l)` (for some `l`) might be consumed by $[\mathtt{l}]_{\mathtt{acq}}$, and thus, might not be framed around $[\mathtt{l}]_{\mathtt{acq}}$ in the inlined program.

Compound statements. It is important to note that our semantic condition requires the *entire body* of an inlined method to be framing, but not necessarily every individual statement in the body. This difference is crucial to capture many realistic methods that contain statements that are not framing, but nevertheless can be inlined in a verification-preserving way. E.g., consider a method whose body contains the following common VeriFast pattern: **open** `[?f]P(x,v); r := x.h;` **close** `[f]P(x,v);` **return** `r`, where `P(x,v)` is a predicate instance with predicate body `x.h` $\mapsto$ `v`, **open** is a ghost operation that exchanges ownership of a predicate instance for ownership of its body, and **close** performs the opposite operation. Here, **open** `[?f]P(x,v)` exchanges a fraction `f` of ownership of `P(x,v)` for `x.h` $\overset{\mathtt{f}}{\mapsto}$ v, where `f` is an existentially-quantified positive fraction. After reading the value of `x.h`, `f` is used to restore the initial ownership of `P(x,v)`. In general, the more ownership of `P(x,v)` in the heap, the higher `f` will be instantiated by VeriFast's heuristic. Thus, **open** `[?f]P(x,v)` is not framing, because ownership of `P(x,v)` cannot be framed around it in general. However, the method body as a whole is framing, and can thus can be captured by the semantic condition.

3.2 Safety Monotonicity

During modular verification, each method execution starts out owning the resources described by its precondition. In contrast, at the same program point in the inlined program, the method owns all resources owned by the caller, which is a superset of those required by any precondition with which the original program verifies modularly. Thus, a statement in the inlined program will typically be verified in a state with more resources than the same statement in the original program.

To ensure that these additional resources do not lead to verification errors (and thereby to non-preserving inlining), our semantic condition requires that successful verification of a statement in some state implies successful verification in any larger state (states with more resources): Statements have to be *safety monotonic*. A statement s is safety monotonic if successful verification of s starting in state φ implies successful verification of s in any *larger state* φ', i.e., if φ' contains at least all the

[4]In general, this requirement applies to all statements that are getting inlined, including loops. Since we focus exclusively on the inlining of method calls here, the call-free statements between calls need not be framing.

```
method client()
{
  a, b := callee()
  { Q(a) }
  l := crLock()
  { P(l,a) }
  acquire(l, a)
  { ... }
}
```

```
method callee(x:Ref):
  (a,b:Ref)
  // requires true
  // ensures Q(a)
{
  a := alloc()
  { Q(a) }
  b := alloc()
  { Q(a) * Q(b) }
}
```

```
method client_inl()
{
  a := alloc()
  { Q(a) }
  b := alloc()
  { Q(a) * Q(b) }
  l := crLock()
  { Q(a) * P(l,b) }
  acquire(l, a) // fails
}
```

Fig. 4. A simplified VeriFast example showing that inlining (with bound 1) can be non-preserving if a call-free statement is not safety monotonic due to VeriFast's heuristic for imprecise assertions. The methods `alloc`, `crLock`, and `acquire` are defined in Fig. 1. Both methods `client` and `callee` verify modularly with the commented-out specification. In particular, because of its specification, method `callee` ensures only `Q(a)` while leaking `Q(b)`, and thus method `client` loses the ownership of `Q(b)` with the call to `callee`, which makes the call to `acquire(l, a)` succeed. However, the inlined version of `client` (shown on the right) does not verify, since the ownership of `Q(b)` is not leaked.

resources in φ and agrees with φ on the common resources (and variables).[5] As we will discuss in Sect. 7, safety monotonicity has been explored in the context of separation logics, but not applied to inlining. Note that framing implies safety monotonicity. Consequently, our semantic condition requires safety monotonicity explicitly only for (potentially compound) call-free statements between calls (which includes the statement after the last method call in the initial statement).

Fig. 4 shows how a violation of safety monotonicity may lead to non-preserving inlining. In this simplified example, `l := crLock(); acquire(l, a)` is *not* safety monotonic, since it verifies in a state with ownership of only `Q(a)`, but fails in a larger state with ownership of both `Q(a)` and `Q(b)`.

3.3 Output Monotonicity

As explained above, a statement s in an inlined program is typically verified in a larger state than in the orginal program. Safety monotonocity ensures that the additional resources do not cause verification of s to fail. However, they could affect the behavior of s such that s removes *more* resources when executed in a larger state and, thereby, causes verification of subsequent statements to fail. Fig. 5 illustrates this problem. Method `callee`'s body is framing, and the `if` statement in method `client` is safety monotonic. Nevertheless, inlining is not verification-preserving because executing the `if` statement in a state with half ownership of `x.f` leaves the state unchanged, whereas executing it in a state with full ownership of `x.f` results in a state with no ownership of `x.f`. This causes verification of the subsequent assignment to `v` in the inlined program to fail.

To avoid this problem, our semantic condition requires statements to be *output monotonic*, in addition to being safety monotonic. A statement that is both safety and output monotonic is called *monotonic*.[6] A statement s is *output monotonic* if executing s in a state φ' that is larger than φ results in a final state that is larger than the state obtained by executing s from φ (assuming s verifies in both states).[7] Output monotonicity constrains the *effect* of a statement on subsequent statements.

[5]Informally, in terms of Hoare triples, s is safety monotonic iff, for all P and R, if the verifier can prove $\{P\}\ s\ \{true\}$ then it can also prove $\{P * R\}\ s\ \{true\}$.

[6]Informally, in terms of Hoare triples, s is monotonic iff, for all P, Q, and R, if the verifier can prove $\{P\}\ s\ \{Q\}$ then it can also prove $\{P * R\}\ s\ \{Q * true\}$.

[7]For non-deterministic programs, one must lift the ordering to sets of states. We ignore this aspect here for simplicity, but show the lifted version in Sect. 4.

```
method client(x:Ref)
  requires x.f ↦ _
  ensures true
{
  { x.f ↦ _ }
  v := callee(x)
  { x.f ↦^0.5 _ }
  if (perm(x.f) >= 1) {
    exhale x.f ↦ _
  }
  { x.f ↦^0.5 _ }
  v := callee(x)
  { true }
}
```

```
method callee(x:Ref)
  : (v:Int)
  // requires x.f ↦^0.5 _
  // ensures true
{
  v := x.f + 1
}
```

```
method client_inl(x:Ref)
  requires x.f ↦ _
  ensures true
{
  { x.f ↦ _ }
  v := x.f + 1
  { x.f ↦ _ }
  if (perm(x.f) >= 1) {
    exhale x.f ↦ _
  }
  { true }
  v := x.f + 1 // fails
}
```

Fig. 5. A simplified example showing that inlining (with bound 1) can be non-preserving if a call-free statement is not output monotonic. The statement **if** (**perm**(x.f) >= 1) { **exhale** x.f ↦ _ } is safety monotonic but not output monotonic. Both methods client and callee verify modularly with the commented-out specification. In particular, because of its specification, method callee leaks ownership of x.f, and thus method client loses half ownership of x.f with each call to callee. Therefore, the if branch is unreachable in the modular verification of method client. However, the inlined version of client (shown on the right) does not verify. Indeed no ownership of x.f is leaked in the inlined program, and thus the if branch is executed, which removes all ownership of x.f. Verification of the line v := x.f + 1 subsequently fails, because some ownership is required to read x.f's value.

It rejects statements that may remove *more* resources when executed in a larger state, thereby causing the verification of subsequent statements to fail. Note that output monotonicity does not subsume safety monotonicity. As an example, the statement **assert perm**(x.f) == 0 is not safety monotonic, since it only verifies in a state with no ownership of x.f. However, this statement is output monotonic, since it does not add or remove any resources. Since framing implies both safety and output monotonicity, our semantic condition requires monotonicity explicitly only for (potentially compound) call-free statements between calls.

The if statement in Fig. 5 is not output monotonic, which causes inlining to be non-preserving. As another example, the statement l := crLock() from Fig. 1 is not output monotonic. Indeed, executing it in a state φ' with ownership of both Q(a) and Q(b) might result in a state with ownership of both Q(a) and P(l,b), while executing it in a state φ with ownership of Q(a) only results in a state with ownership of P(l,a). While φ' is a state with more resources than φ, the resulting states are not comparable, which violates output monotonicity.

Practical use cases. While any framing statement is monotonic, the converse does not hold. In particular, a number of useful patterns are monotonic but not framing. One example is the statement **exhale** x.f $\overset{\texttt{perm(x.f)}}{\mapsto}$ _, which releases all ownership to x.f held by the method execution. It is monotonic since it always verifies and the resulting state contains no permission to x.f, but it is not framing because permission to x.f cannot be framed around it. A similar statement is used in RSL-Viper to transfer resources under a modality. The statement **open** P in VeriFast, where P is a predicate, behaves similarly, as explained in Sect. 3.1. Even though the statement **open** P is monotonic, it is not framing, since, in general, the more ownership of P is held, the more ownership of P is exchanged, and thus ownership of P cannot be framed around this statement. Another example of a monotonic statement that is not framing is releasing some existentially-quantified fractional ownership of a resource (wildcard in Viper, dummy or existential fraction in VeriFast), e.g., when calling a trusted library function that requires read access to a heap location.

3.4 Bounded Relaxations

Requiring (1) inlined method bodies to be framing and (2) call-free statements between calls to be monotonic is sufficient to guarantee verification-preserving inlining, but overly restrictive. The framing and (safety and output) monotonicity properties presented so far quantify over two *arbitrary* states $\varphi \preceq \varphi'$. These properties consider *arbitrary* executions of a statement s, instead of considering what resources the inlined and original programs may actually own before executing s.

The example from Fig. 1 illustrates why this condition is too restrictive. The statement $s \triangleq$ (`l := crLock()`) is not monotonic (and thus not framing), as explained in Sect. 3.3. However, assume that we remove `b := alloc()` from the example. In this case, inlining is verification-preserving, since the heuristic can instantiate `Q(?x)` with `Q(a)` only. Nevertheless, our monotonicity requirement rejects this program, since s is not output monotonic. s violates output monotonicity when φ' owns both `Q(a)` and `Q(b)` but φ owns `Q(a)` only. This violation is irrelevant in our modified example, since s is only executed in states that own at most `Q(a)` (in both the inlined and the original program).

Bounded properties. To take into account which states can actually occur in executions of the inlined and the original program, our semantic condition requires only bounded relaxations of our framing and monotonicity properties. For output monotonicity, the bounded version is parameterized by a set of states T (the *resource bound*) and restricts φ, φ' to be smaller than at least one state in T. The resource bound is set to the possible program states in the inlined program at the relevant point. The bounded relaxations of framing and safety monotonicity are analogous.

In our modified example, the statement `l := crLock()` is bounded output monotonic w.r.t. the inlined program state ϕ_p before this statement, since ϕ_p owns `Q(a)` only, and thus, φ and φ' over which the condition quantifies cannot own more. In the following, when we refer to safety monotonicity, output monotonicity, or framing, we mean the bounded relaxations. We explain how to automatically check these properties in Sect. 5.

Practical use cases. The bounded relaxation is crucial for VeriFast and GRASShopper, as we show in Sect. 6. Many methods in their test suites contain existentially-quantified (and thus imprecise) assertions. Without the relaxation, none of these methods would satisfy the semantic condition, even though many of them can be inlined in a verification-preserving way in caller contexts where the existential quantifications are unambiguous, as in our modified example.

The bounded relaxation is also crucial for Nagini. Python allows one to create object fields dynamically. Nagini encodes the Python assignment `x.f = v` into Viper as follows, where `P(x,f)` represents the permission to create the field `f` of `x`: **if** `(`**perm**`(P(x,f)) > 0) {` **exhale** `P(x,f);` **inhale** `x.f` $\mapsto$ `_ }; x.f := v`. This encoding replaces the resource `P(x,f)`, if available, with ownership of the field. While this encoding is not unbounded safety monotonic, it is always bounded safety monotonic (and also framing), because Nagini ensures that `x.f` $\mapsto$ _ and any ownership of `P(x,f)` are mutually exclusive and hence no state in the bound T contains both. Intuitively, Nagini's proof search heuristics never has a choice which resource to use and, thus, cannot err.

3.5 Inlining Partial Annotations

We conclude this section by first showing how we inline calls with partial annotations (i.e., where a subset of methods have annotations that may themselves be incomplete) and then explaining how to generalize the notion of verification-preserving inlining in the presence of partial annotations.

```
method b(x,y:Ref)
  requires P(x,v) ∗ y.h ↦ _
  ensures P(x,v) ∗ y.h ↦ v
{ c(x,y) }
```

```
method c(x,y:Ref)
  requires [?f]P(x,v)
  ensures [f]P(x,v)
{ open [f]P(x,v)
  y.h := x.h + 1
  close [f]P(x,v) }
```

```
assert [?f]P(x,v)
open [f]P(x,v)
y.h := x.h + 1
close [f]P(x,v)
assert [f]P(x,v)
```

Fig. 6. A VeriFast example showing *verification-preserving* inlining in the presence of ghost code and partial annotations. c's ghost code (in red) requires c's partial specification (in blue) to bind the existential parameter f. Predicate P(x,v) is a predicate instance with predicate body x.h ↦ v. The snippet on the very right shows the inlined body of b when c's specification and ghost code are included.

Verification-preserving inlining with partial annotations. As we explained in Sect. 1, inlining is a useful stepping stone toward modular verification, since it allows one to detect errors before adding annotations and to validate partial annotations that arise during the iterative process of annotating methods (e.g., by iteratively adding conjuncts to pre- and postconditions). A partial annotation is an annotation that may not yet contain enough information in order for modular verification to succeed. Verifying callees modularly with partial annotations may fail, e.g., because the callee's precondition does not provide all resources needed to verify its body. Therefore, inlining with partial annotations still reasons about calls by replacing them by the body of the callee method. Nevertheless, in order to validate partial annotations, inlining proves that they actually hold by asserting them in the inlined program. More precisely, whenever a call to a method m with a partial annotation is inlined, the inlined program *asserts* m's precondition, then executes m's body, and finally *asserts* m's postcondition. Asserting the conditions checks that the resources they describe are held by the current method execution, but does not add or remove any resources.

The definition of verification-preserving inlining is adjusted accordingly in the presence of partial annotations. Inlining a program with bound n and partial annotations $\mathcal{A}$ is verification-preserving if the following holds: If the program verifies modularly w.r.t. some annotation *that is more complete than* $\mathcal{A}$ (i.e., all method pre- and postconditions are stronger than the corresponding pre- and postconditions in $\mathcal{A}$), then the program inlined with bound n and partial annotations $\mathcal{A}$ verifies. Thus, if inlining with $\mathcal{A}$ is verification-preserving for a program, then an error in the inlined program implies that the original program cannot be verified modularly for any annotation that is more complete than $\mathcal{A}$ (e.g., no conjuncts can be added to $\mathcal{A}$ to make the original program verify).

We also adjust the semantic condition to take partial annotations into account, by first applying a syntactic transformation on the program that asserts the partial annotations before and after method calls (we make this precise in the next section).

Example. Consider the example showing verification-preserving inlining in Fig. 6, which includes a partial annotation for method c and ghost code in c's method body. This example shows a scenario where (1) verification-preserving inlining can be used to find errors with partial annotations, and (2) inlining the method body makes sense only if one takes partial annotations into account. By asserting partial annotations in the inlined program, our technique handles both aspects.

This example is based on the common VeriFast pattern already described in Sect. 3.1. The existential quantification over f in c's specification enables more possible callers and transfers back the initial ownership. The ghost operations **open** [f]P(x,v) and **close** [f]P(x,v) are required, since the verifier does not automatically unroll P(x,v) to justify reading x.h (the predicate body of P(x,v) is x.h ↦ v). These operations have a meaning only due to c's precondition that binds f, which shows that we need to inline partial annotations. Note that `c`'s specification is truly partial, since ownership of `y.h` would be required to justify the assignment in a modular proof.

The inlined body of `b` with its specification and ghost code is shown on the right of Fig. 6. Asserting the precondition `[?f]P(x,v)` checks whether some ownership of `P(x,v)` is held and `f` binds the fractional ownership amount that VeriFast picks to prove the assertion. In this case, VeriFast's heuristic binds `f` to the currently-owned fraction of `P(x,v)`, i.e., to 1.

The inlined program fails, since `y.h` does not hold the same value as `x.h`, which is required by `b`'s postcondition. Since there is no specification for `c` that can make `b` verify, inlining is verification-preserving and thus, inlining detects a true error without the user having to provide ownership of `y.h` in `c`'s specification. The semantic condition holds in this example, because the inlined body of `c` (including the **`assert`** statements) *as a whole* satisfies the frame rule.

4 VERIFICATION-PRESERVING INLINING

In Sect. 3, we motivated the building blocks of the semantic condition. In this section, we formally define the semantic condition and prove that inlining is verification-preserving when the semantic condition holds. In order to express this formal result in a general way, we define a parametric verification language that captures the essence of verification languages such as GRASShopper, VeriFast, and Viper. To capture different models of resources, the states of this language are elements of a separation algebra. We formalize inlining and the semantic condition for this language, and prove that inlining is verification-preserving under the semantic condition (Theorem 4.5). We first consider a version of inlining that ignores annotations, and then show how to leverage this version to support inlining with partial annotations. All results presented in this section have been mechanized in Isabelle/HOL [Dardinier et al. 2023]. As explained in Sect. 3, this section focuses on methods calls, but loops are handled in App. A and App. B of the TR, and in the mechanization.

4.1 State Model and Verification Language

We present the essential aspects of our verification language here; additional formal definitions are given in App. A of the TR.

State model. To reflect the *verifier semantics* (see Sect. 3), our state model contains separation logic resources in addition to a standard state with local variables and a heap. The models of verifiers such as GRASShopper, VeriFast, or Viper are essentially captured by a separation algebra [Calcagno et al. 2007; Dockins et al. 2009] where Σ is the set of states, $\oplus : \Sigma \times \Sigma \rightarrow \Sigma$ is a partial operation that is commutative and associative, and $u \in \Sigma$ is a neutral element for $\oplus$. Intuitively, two states can be added if they agree on the values of common local variables and heap locations and if their resources can be combined in a consistent way. The addition then contains the union of their local variables and heap values, and the combination of their resources. We write $\varphi \# \varphi'$ if $\varphi \oplus \varphi'$ is defined, and $\varphi' \preceq \varphi \iff (\exists \varphi'' \in \Sigma.\ \varphi = \varphi' \oplus \varphi'')$. We lift the operators $\oplus$ and $\preceq$ to sets of states T and U, where $T \oplus U \triangleq \{\varphi \oplus \varphi' | \varphi \in T \land \varphi' \in U \land \varphi \# \varphi'\}$ and $U \preceq T \iff (\forall \varphi \in T.\ \exists \varphi' \in U.\ \varphi' \preceq \varphi)$.

Language and semantics. We consider a parametric verification language with the previously-described state model and the following statements:

$$S ::= S; S \mid \textbf{if } (*) \ \{S\} \textbf{ else } \{S\} \mid \textbf{while } (A) \ \{S\} \mid \overrightarrow{V} := m(\overrightarrow{V}) \mid \textbf{skip}$$
$$\mid \textbf{assume } A \mid \textbf{assert } A \mid \textbf{inhale } A \mid \textbf{exhale } A \mid \textbf{var } \overrightarrow{V} \mid \textbf{havoc } \overrightarrow{V} \mid \textbf{custom } O$$

where A represents assertions and $\overrightarrow{V}$ lists of variable identifiers. O is a parameter of the language used to represent verifier-specific statements, such as **open** and **`close`** in VeriFast.

Most non-custom statements of this language are standard and have the usual semantics. Our `if` statement is non-deterministic and can model both non-deterministic choice and (by using **assume** statements in the branches) deterministic branching. As explained earlier, **inhale** A combines the current verification state with a state satisfying A, while **exhale** A removes a state satisfying A from the current verification state (and fails if this is not possible). During verification of **inhale** A, the verifier must consider *all possible* states that satisfy A. However, for **exhale** A it can *choose* how to satisfy A, for instance, how to instantiate an existential quantifier. This choice is embodied by a heuristic, which is a parameter of our verifier semantics. The only assumptions we make about the verifier's heuristics is that they are *local* and *deterministic*, i.e., the choices made by the verifier are fully determined by the current verification state. For instance, for the same verification state, a heuristic will always make the same choice for an existential quantifier. This is the case for verifiers such as GRASSHopper, VeriFast, and verifiers built on top of Viper. In contrast, Caper uses backtracking, which is not local but depends on the whole program.

For an annotation $\mathcal{A}$ containing pre- and postconditions for every method (transitively) called from the statement s, we write $ver_{\mathcal{A}}(\varphi, s)$ if s verifies modularly for executions starting in the initial state φ, where method calls are verified using only their pre- and postconditions in $\mathcal{A}$. In this case, we define $sem_{\mathcal{A}}(\varphi, s)$ as the set of states that are reached after executing s in the state φ w.r.t. $\mathcal{A}$. Note that $ver_{\mathcal{A}}(\varphi, s)$ and $sem_{\mathcal{A}}(\varphi, s)$ model the verifier semantics (as introduced in Sect. 3).

4.2 Inlining without Annotations

We now formally define inlining and the semantic condition for our language, and then prove that the latter implies verification-preserving inlining. We ignore annotations here, but they are handled in Sect. 4.3. While inlining, we need to keep track of the already-used variables, to avoid variable capturing. For simplicity, we ignore all renaming issues here, but our Isabelle/HOL formalization covers this aspect. The inlining function $inl^n_M(s)$ yields the statement s where all calls to methods from M are substituted by their bodies up to the inlining bound n (annotations are ignored):

Definition 4.1. **Inlining (ignoring renaming issues and loops).**

$$inl^n_M(s) \triangleq s \text{ (if } s \text{ is call-free)} \qquad inl^0_M(\vec{y} := m(\vec{x})) \triangleq \textbf{assume } \bot$$

$$inl^n_M(s_1; s_2) \triangleq inl^n_M(s_1);\, inl^n_M(s_2) \qquad inl^{n+1}_M(\vec{y} := m(\vec{x})) \triangleq inl^n_M(s_m)$$

$$inl^n_M(\textbf{if } (*)\ \{s_1\} \textbf{ else } \{s_2\}) \triangleq \textbf{if } (*)\ \{inl^n_M(s_1)\} \textbf{ else } \{inl^n_M(s_2)\}$$

where s_m is the body of method $m \in M$ with arguments correctly substituted.

When the inlining bound n has reached 0, additional calls render the execution infeasible. Otherwise, a method call is replaced by the method body (with suitable substitutions, omitted here).

Monotonicity and framing. Before we show the semantic condition, we formalize its key building blocks. To specify the bounded relaxation (motivated in Sect. 3.4), we define the shorthand $(\varphi \ll T) \triangleq (\exists \varphi' \in T.\, \varphi \preceq \varphi')$. The following definition combines the bounded safety and output monotonicity properties motivated in Sect. 3.2 and Sect. 3.3:

Definition 4.2. **Bounded safety and output monotonicity.**

$$\begin{aligned} mono_{\mathcal{A}}(T, s) \triangleq (\forall \varphi_1, \varphi_2 \in \Sigma.\, \varphi_1 \preceq \varphi_2 \ll T \land ver_{\mathcal{A}}(\varphi_1, s) & \\ \implies ver_{\mathcal{A}}(\varphi_2, s) \land & \quad \text{(safety monotonicity)} \\ sem_{\mathcal{A}}(\varphi_1, s) \preceq sem_{\mathcal{A}}(\varphi_2, s)) & \quad \text{(output monotonicity)} \end{aligned}$$

$mono_{\mathcal{A}}(T,s)$ states that if s verifies in a state φ_1, then s also verifies in a larger state φ_2 (safety monotonicity), and executing s in φ_2 results in a larger set of states than executing s in φ_1 (output monotonicity). $\varphi_2 \ll T$ expresses that the larger state φ_2 is smaller than at least one state of T.

The following definition captures the bounded framing property motivated in Sect. 3.1:

Definition 4.3. **Bounded framing.**[8]

$$framing_{\mathcal{A}}(T,s) \triangleq (\forall \varphi, r \in \Sigma.\ \varphi \# r \wedge \varphi \oplus r \ll T \wedge ver_{\mathcal{A}}(\varphi, s) \Longrightarrow ver_{\mathcal{A}}(\varphi \oplus r, s) \wedge sem_{\mathcal{A}}(\varphi, s) \oplus \{r\} \preceq sem_{\mathcal{A}}(\varphi \oplus r, s))$$

$framing_{\mathcal{A}}(T,s)$ holds iff, for any state (smaller than some state in T) that can be decomposed into $\varphi \oplus r$ s.t. s verifies in φ, it holds that executing s in the state $\varphi \oplus r$ verifies and results in a larger[9] set of states than executing s in φ and adding the frame r afterwards (recall that we have lifted the $\oplus$ and $\preceq$ operators to sets of states). As an example, the statement **exhale** `x.f` $\overset{\texttt{perm(x.f)}}{\mapsto}$ _ is framing if and only if no state in T contains non-zero ownership of `x.f`. Otherwise, we can prove that it is not framing, by choosing a frame r with non-zero ownership of `x.f`.

Semantic condition. For an inlining bound n, a set of methods M, a set of states T, and a statement s, we denote our semantic condition by $SC^n_M(T,s)$, and define it as follows (where the set of states T is the bound we use for mono and framing):

Definition 4.4. **Semantic condition (ignoring renaming issues and loops).**

$$SC^n_M(T,s) \triangleq mono_\epsilon(T,s) \qquad \text{(if } s \text{ is call-free)}$$
$$SC^n_M(T, s_1; s_2) \triangleq SC^n_M(T,s_1) \wedge SC^n_M(\overline{sem}_\epsilon(T, inl^n_M(s_1)), s_2)$$
$$SC^n_M(T, \textbf{if } (*)\ \{s_1\} \textbf{ else } \{s_2\}) \triangleq SC^n_M(T,s_1) \wedge SC^n_M(T,s_2)$$
$$SC^0_M(T, \vec{y} := m(\vec{x})) \triangleq \top$$
$$SC^{n+1}_M(T, \vec{y} := m(\vec{x})) \triangleq framing_\epsilon(T, inl^n_M(s_m)) \wedge SC^n_M(T, s_m)$$

where ϵ is the empty annotation, s_m is the body of method $m \in M$ with arguments correctly substituted, and $\overline{sem}_{\mathcal{A}}(T,s) \triangleq \bigcup_{\varphi \in \Sigma | (\exists \varphi' \in T. \varphi \preceq \varphi') \wedge ver_{\mathcal{A}}(\varphi,s)} (sem_{\mathcal{A}}(\varphi, s))$.

As explained in Sect. 3, we require call-free statements to be mono, and inlined method bodies to be framing. For the sequential composition, we need the auxiliary function $\overline{sem}$, which applies the *sem* function to all states φ in which s verifies and that is smaller than some state in T. This auxiliary function is required to compute the right resource bound for the framing and monotonicity properties, in order to ensure verification-preserving inlining. Note that the semantic condition does not depend on any annotation, since mono and framing are enforced only on call-free statements.

Verification-preserving inlining. Using inlining and the semantic condition, we can express and prove the following theorem (under some standard well-formedness conditions).

THEOREM 4.5. ***Verification-preserving inlining.*** *For any well-formed program (s, M) for which $SC^n_M(\{u\}, s)$[10] holds, if there exists an annotation $\mathcal{A}$ for M such that:*
(1) all methods in M verify modularly w.r.t. $\mathcal{A}$, and

[8] For the sake of presentation, we ignore the (usual) side condition that r does not contain variables modified by the statement s, but this is handled in our ISABELLE/HOL formalization.
[9] It would also be correct to require $sem_{\mathcal{A}}(\varphi \oplus r, s) = sem_{\mathcal{A}}(\varphi, s) \oplus \{r\}$ instead of $sem_{\mathcal{A}}(\varphi, s) \oplus \{r\} \preceq sem_{\mathcal{A}}(\varphi \oplus r, s)$. However, there are cases where having more resources available leads to more resources being generated by a statement (e.g., changing the modality of a resource in RSL-Viper), and these cases are captured only by the latter (weaker) requirement.
[10] Recall that u is the neutral element of $\oplus$, that is, the empty state.

(2) the initial statement s verifies modularly w.r.t. $\mathcal{A}$, that is, $ver_{\mathcal{A}}(u, s)$,
then the program (s, M) inlined with the inlining bound n verifies: $ver_{\epsilon}(u, inl^n_M(s))$.

Proof sketch. We prove the following invariant relating the original and the inlined program (assuming the semantic condition holds and the original program verifies modularly): The verification state of the inlined program has at least as many resources as the corresponding verification state during modular verification of the original program. Formally, we prove that, for any two states $\varphi \preceq \varphi'$ smaller than some state in the inlined program at the relevant point (i.e., the resource bound), if $ver_{\mathcal{A}}(\varphi, s)$ holds, then $ver_{\epsilon}(\varphi', inl^n_M(s))$ and $sem_{\mathcal{A}}(\varphi, s) \preceq sem_{\epsilon}(\varphi', inl^n_M(s))$ hold. We prove this invariant to hold before and after every method call (and every loop iteration), by induction on the structure of the inlined program. In the case where we inline a method call, we know that this method call has been modularly verified using the frame rule. We use the fact that the inlined method body is framing (from the semantic condition) to mimic the application of the frame rule for the inlined program. In the case of a call-free program statement, we use the fact that this call-free statement is monotonic to prove that it preserves the aforementioned invariant. □

Since inlining and the semantic condition do not depend on any annotation, this theorem implies the following result, which we are mostly interested in and have proved in Isabelle/HOL: If the verification of the inlined program fails and the semantic condition holds, then there does not exist an annotation such that the original program verifies modularly w.r.t. this annotation. In other words, any error in the inlined program corresponds to a true error in the original program.

4.3 Inlining with Partial Annotations

We extend the formalization to handle partially-annotated programs in two steps: We first apply a syntactic transformation *assertAnnot* on the program that adds **assert** statements to check method specifications, and then inline the resulting program using the annotation-agnostic *inl* function defined in Sect. 4.2. This two-step approach allows us to leverage our previous results to prove that the semantic condition guarantees that inlining with partial annotations is verification-preserving (Theorem 4.7), which we have also formalized and proved in Isabelle/HOL [Dardinier et al. 2023].

Definition 4.6. **The *assertAnnot* syntactic transformation.**
Let (s, M) be a program with an annotation $\mathcal{A}$. $assertAnnot_{\mathcal{A}}(s, M)$ returns the program $(assertAnnotStmt_{\mathcal{A}}(s), assertAnnotMethods_{\mathcal{A}}(M))$. $assertAnnotStmt_{\mathcal{A}}(s)$ asserts the method precondition (resp. postcondition) before (resp. after) each method call in s. In particular, $assertAnnotStmt_{\mathcal{A}}(\overrightarrow{y} := m(\overrightarrow{x})) \triangleq$ **assert** $P * \text{true}$; $\overrightarrow{y} := m(\overrightarrow{x})$; **assert** $Q * \text{true}$ where P (resp. Q) is method m's precondition (resp. postcondition) in $\mathcal{A}$ with the arguments correctly substituted. $assertAnnotMethods(M)$ returns the same methods as M, but where the method body s_m of $m \in M$ is modified to check the pre- and postcondition of m and all methods it calls: **assert** $P * \text{true}$; $assertAnnotStmt_M(s_m)$; **assert** $Q * \text{true}$, where P (resp. Q) is m's precondition (resp. postcondition) in $\mathcal{A}$ with the arguments correctly substituted.

Given a program (s, M), a bound n, and an annotation $\mathcal{A}$, we define its inlined version *with partial annotations* as $inl^n_{M_{\mathcal{A}}}(s_{\mathcal{A}})$ where $(s_{\mathcal{A}}, M_{\mathcal{A}}) = assertAnnot_{\mathcal{A}}(s, M)$. For each call, the resulting inlined program asserts the precondition right before the call and also at the beginning of the callee (analogously for the postcondition). While it seems redundant to assert the precondition twice in the inlined program, the second assertion right at the beginning of the callee makes the semantic condition defined in Sect. 4.2 more precise: The assertion forces properties on the inlined method body to only take into account states at the beginning of the body that satisfy the precondition (and analogously for the postcondition). Note that conjoining the precondition (resp. postcondition) with true in *assertAnnot* is crucial for verifiers based on classical SL (such as GRASShopper), because for calls one must check that the caller context has *at least* the resources specified by the precondition (resp. postcondition) before the call (resp. after the call).

We can now state and prove the following theorem:

Theorem 4.7. ***Verification-preserving inlining with partial annotations.*** *Let (s, M) be a well-formed program with annotations $\mathcal{A}$ and $\mathcal{B}$ s.t. $\mathcal{B}$ is more complete than $\mathcal{A}$ (i.e., all method pre- and postconditions in $\mathcal{B}$ are logically stronger than the corresponding pre- and postconditions in $\mathcal{A}$), and let $(s_{\mathcal{A}}, M_{\mathcal{A}}) = assertAnnot_{\mathcal{A}}(s, M)$. If*

(1) $SC_{M_{\mathcal{A}}}(\{u\}, s_{\mathcal{A}})$ holds, and

(2) all methods in M verify modularly w.r.t. $\mathcal{B}$, and

(3) the initial statement s verifies modularly w.r.t. $\mathcal{B}$ that is, $ver_{\mathcal{B}}(u, s)$,

then the program $(s_{\mathcal{A}}, M_{\mathcal{A}})$ inlined with annotation $\mathcal{A}$ and inlining bound n verifies: $ver_{\epsilon}(u, inl^n_{M_{\mathcal{A}}}(s_{\mathcal{A}}))$.

Proof sketch. Let $(s_{\mathcal{B}}, M_{\mathcal{B}}) \triangleq assertAnnot_{\mathcal{B}}(s, M)$. Using (2) and (3), we prove that $(s_{\mathcal{B}}, M_{\mathcal{B}})$ verifies modularly w.r.t. $\mathcal{B}$: This holds because the additional assertions in $(s_{\mathcal{B}}, M_{\mathcal{B}})$ reflect what must hold before and after each call when modularly verifying w.r.t. $\mathcal{B}$. Moreover, using (1) and the fact that $\mathcal{B}$ is more complete than $\mathcal{A}$, we prove that $SC_{M_{\mathcal{B}}}(\{u\}, s_{\mathcal{B}})$ holds (using that $(s_{\mathcal{A}}, M_{\mathcal{A}})$ and $(s_{\mathcal{B}}, M_{\mathcal{B}})$ differ only in their **assert** statements). We can now apply Theorem 4.5 to $(s_{\mathcal{B}}, M_{\mathcal{B}})$ with annotation $\mathcal{B}$, which gives us $ver_{\epsilon}(u, inl^n_{M_{\mathcal{B}}}(s_{\mathcal{B}}))$. This implies $ver_{\epsilon}(u, inl^n_{M_{\mathcal{A}}}(s_{\mathcal{A}}))$, because the statements $inl^n_{M_{\mathcal{B}}}(s_{\mathcal{B}})$ and $inl^n_{M_{\mathcal{A}}}(s_{\mathcal{A}}))$ differ only in their **assert** statements, and, since $\mathcal{B}$ is more complete than $\mathcal{A}$, successful verification of the **assert** statements in $inl^n_{M_{\mathcal{B}}}(s_{\mathcal{B}})$ implies successful verification of the **assert** statements in $inl^n_{M_{\mathcal{A}}}(s_{\mathcal{A}})$. □

Similarly to Theorem 4.5, we are particularly interested in the following corollary, which we have proved in Isabelle/HOL: If $SC_{M_{\mathcal{A}}}(\{u\}, s_{\mathcal{A}})$ holds, and verification of $inl^n_{M_{\mathcal{A}}}(s_{\mathcal{A}})$ fails, then there does not exist an annotation $\mathcal{B}$ more complete than $\mathcal{A}$ such that (s, M) verifies modularly w.r.t. $\mathcal{B}$. That is, there is no way to complete the partial annotation $\mathcal{A}$ (e.g., by adding conjuncts to pre- or postconditions) such that the program verifies modularly.

5 AUTOMATION FOR VERIFICATION-PRESERVING INLINING

Theorems 4.5 and 4.7 from Sect. 4 state that errors in the inlined program correspond to true errors in the original program, *provided* that the semantic condition holds for this program and the inlining bound. While inlining (Def. 4.1) and the syntactic transformation (Def. 4.6) are straightforward to implement, checking the semantic condition directly is challenging. Both the mono and framing properties are hyperproperties [Clarkson and Schneider 2008] (properties of multiple executions) that combine universal and existential quantification over states. Automatic program verifiers can check properties *for all* executions, but cannot reason about the existence of executions. To work around this limitation, we present two conservative approximations of the semantic condition that can be checked syntactically and with a standard program verifier, respectively.

Syntactic condition. We first provide *syntactic* versions of mono and framing that are fast and easy to check, to quickly accept programs that do not use features that could lead to non-preserving inlining. A program is syntactic mono (resp. framing) if the program does not contain any *syntactic* features that could be the reason for violating mono (resp. framing). Such *violating features* include operations that inspect the resources held in a state (e.g., the **perm** feature in Viper). Violating features also include any feature that could trigger proof search strategies for imprecise assertions. In GRASShopper, VeriFast, and Viper, this includes, for instance, imprecise assertions in preconditions of library methods. The syntactic check overapproximates the imprecise assertions by

```
assume perm(x.f) ≤ 1/2;                exist := exist ∧ perm(x.f) ≤ 1/2;
assert perm(x.f) ≥ 1/4                 if (exist) { assume perm(x.f) ≥ 1/4 }
```

Fig. 7. The statement s (sequential composition of the two statements on the left) is mono and framing, but rejected by the syntactic rules since it uses resource introspection. Our structural rules admit this statement. The proof obligation used to check these rules, $guardExecs(s, exist)$, is shown on the right.

checking for components such as existentially-quantified parameters that could be the cause of imprecision. In App. E of the TR, we provide details about violating features for the three verifiers.

These syntactic checks are useful to quickly identify programs for which inlining is clearly verification-preserving, but are too coarse to validate non-trivial applications of advanced features (including the examples in Sect. 2). For example, the statement on the left of Fig. 7 is mono and framing, but is rejected by the syntactic check since it uses Viper's **perm** feature, which can be used to encode proof search strategies that might lead to non-preserving inlining, as shown in Sect. 2.

Structural condition. To validate more complex programs, we also provide *structural* versions of mono and framing that are more precise than the syntactic versions and that can nevertheless be checked by a standard program verifier. For simplicity, the rest of this section focuses on the structural version of mono, but the treatment of framing is analogous (see App. F of the TR).

The structural mono property strengthens the mono property (Def. 4.2) such that it can be automatically checked via a program verifier. Below, we show its definition for an annotation $\mathcal{A}$, a set of states T, and a statement s. In this definition, the *determinization function det* (which maps three states and a statement to a set of states) corresponds to a non-empty subset of $sem_{\mathcal{A}}(\varphi_1, s)$, obtained via the process of *determinization* (explained later in this section). In the case of a deterministic statement s, $det(\varphi_1, \varphi_2, \varphi_2', s) = sem_{\mathcal{A}}(\varphi_1, s)$, since $sem_{\mathcal{A}}(\varphi_1, s)$ contains at most one element, and thus it is the only subset of itself that might be non-empty.

Definition 5.1. **Structural mono**

$$
\begin{aligned}
&structMono_{\mathcal{A}}(T, s) \triangleq \forall \varphi_1, \varphi_2 \in \Sigma.\ \varphi_1 \preceq \varphi_2 \ll T \land ver_{\mathcal{A}}(\varphi_1, s) \Longrightarrow \\
&\quad ver_{\mathcal{A}}(\varphi_2, s) \land && \text{(safety monotonicity)} \\
&\quad \begin{pmatrix} \forall \varphi_2' \in sem_{\mathcal{A}}(\varphi_2, s).\ (\forall \varphi_1' \in det(\varphi_1, \varphi_2, \varphi_2', s).\ \varphi_1' \preceq \varphi_2') \land \\ \emptyset \subset det(\varphi_1, \varphi_2, \varphi_2', s) \subseteq sem_{\mathcal{A}}(\varphi_1, s) \end{pmatrix} && \text{(structural output mono)}
\end{aligned}
$$

The structure of the structural mono definition and the mono definition (Def. 4.2) are identical. The definitions differ only in the conjunct for output monotonicity. In the original mono definition, this conjunct is given by $sem_{\mathcal{A}}(\varphi_1, s) \preceq sem_{\mathcal{A}}(\varphi_2, s)$, which, after expanding $\preceq$ (see definition of $\preceq$ in Sect. 4.1), is equivalent to $\forall \varphi_2' \in sem_{\mathcal{A}}(\varphi_2, s).\ \exists \varphi_1' \in sem_{\mathcal{A}}(\varphi_1, s).\ \varphi_1' \preceq \varphi_2'$. This formula contains an existential quantifier over states that is nested within a universal quantifier, thus making it hard to automatically reason about.[11] The structural mono definition circumvents this forall-exists alternation issue by strengthening the existential quantifier over φ_1' to a universal quantifier over a non-empty range. That is, we replace the existentially-quantified formula $\exists \varphi_1' \in sem_{\mathcal{A}}(\varphi_1, s).\ \varphi_1' \preceq \varphi_2'$ from the original definition of mono by the universally-quantified $\forall \varphi_1' \in det(\varphi_1, \varphi_2, \varphi_2', s).\ \varphi_1' \preceq \varphi_2'$ (which we call the *universal determinization condition*) and the additional requirement that the range $det(\varphi_1, \varphi_2, \varphi_2', s)$ is a non-empty subset of $sem_{\mathcal{A}}(\varphi_1, s)$.

It is easy to show that structural mono implies mono, as we have proved in Isabelle/HOL:

Lemma 5.2. ***Structural mono implies mono:*** $structMono_{\mathcal{A}}(T, s) \Longrightarrow mono_{\mathcal{A}}(T, s)$.

[11]Note that the existential quantifier hidden in $\varphi_2 \ll T$ in Def. 5.1 is not problematic for automation because it occurs on the left-hand side of an implication and is, thus, equivalent to a top-level universal quantifier.

Proof sketch. By the similar structure between the definitions of mono (Def. 4.2) and structural mono (Def. 5.1), we simply need to show that (a) $\forall \varphi_2' \in sem_{\mathcal{A}}(\varphi_2, s).\ (\forall \varphi_1' \in det(\varphi_1, \varphi_2, \varphi_2', s).\ \varphi_1' \preceq \varphi_2') \wedge \emptyset \subset det(\varphi_1, \varphi_2, \varphi_2', s) \subseteq sem_{\mathcal{A}}(\varphi_1, s)$ implies (b) $sem_{\mathcal{A}}(\varphi_1, s) \preceq sem_{\mathcal{A}}(\varphi_2, s)$. (b) is equivalent, by definition (Sect. 4.1), to $\forall \varphi_2' \in sem_{\mathcal{A}}(\varphi_2, s).\ \exists \varphi_1' \in sem_{\mathcal{A}}(\varphi_1, s).\ \varphi_1' \preceq \varphi_2'$. We assume (a), and want to show (b). Let $\varphi_2' \in sem_{\mathcal{A}}(\varphi_2, s)$. From (a), we know that $det(\varphi_1, \varphi_2, \varphi_2', s)$ is not empty. Thus, let φ_1' be any state from $det(\varphi_1, \varphi_2, \varphi_2', s)$. Then, from (a), φ_1' also belongs to $sem_{\mathcal{A}}(\varphi_1, s)$, and $\varphi_1' \preceq \varphi_2'$ holds, which proves (b), and thus concludes the proof. □

The structural framing property is obtained analogously by modifying the framing definition (Def. 4.3). It is easy to show that structural framing implies framing (see App. F of the TR). The structural condition $StructC^n_M(T, s)$ is defined identically to the semantic condition $SC^n_M(T, s)$ (Def. 4.4), except that the mono and framing properties are replaced by the corresponding structural properties. We proved in Isabelle/HOL that the structural condition implies the semantic condition:

Theorem 5.3. $StructC^n_M(T, s) \Longrightarrow SC^n_M(T, s)$

Proof sketch. Since $StructC^n_M(T, s)$ is defined identically to $SC^n_M(T, s)$ except that structural mono (resp. stuctural framing) is used instead of mono (resp. framing), this statement follows immediately by induction on $StructC^n_M(T, s)$ using that structural mono implies mono (Lemma 5.2) and structural framing implies framing (Lemma F.2 in App. F of the TR). □

Automating the structural condition. In the following, we explain how we check the structural condition *automatically*, which boils down to automatically checking structural mono and structural framing. Our approach is implemented by emitting additional proof obligations in Viper's verification condition generator, which builds on the Boogie verifier [Leino 2008]. For the sake of concreteness, we will explain these additional proof obligations in terms of this implementation. However, they can be generated in any verifier that can (a) express an ordering on states and (b) non-deterministically choose a state smaller than some other state. Both requirements are met by the prevalent implementation techniques for automatic SL verifiers: verification condition generation and symbolic execution. In our implementation, we satisfy both requirements by relying on the total-heap representation of SL states [Parkinson and Summers 2012] used by Viper's verification condition generator where a state consists of a heap, a permission mask (mapping resources to the held ownership amounts), and a store of local variables. The heap and the mask are represented in Boogie with maps. This representation allows us to satisfy requirement (a) by universally quantifying over the contents of the maps representing the heaps and the masks of both states (e.g., to express that one mask contains pointwise less permission than the other), and requirement (b) by picking fresh maps and then constraining them suitably via **assume** statements.

Given these two requirements, we can express the proof obligations for structural mono and framing. Both are hyperproperties because they relate two executions of the statement s. As is common, we use self-composition [Barthe et al. 2011] to reduce these hyperproperties to trace properties that can be checked by a standard verifier such as Boogie.

We now explain the proof obligations for the structural mono property for a statement s. Checking structural framing is analogous as explained in App. F of the TR. We show the simpler case where s is deterministic. In this case, the determinization function is given by $det(\varphi_1, \varphi_2, \varphi_2', s) = sem_{\mathcal{A}}(\varphi_1, s)$. At the end of this section, we will then explain how to handle non-deterministic statements.

Our structural condition requires the call-free statements between method calls to be structurally mono. To check this property, we precede each such statement s with the code shown in Fig. 8, which generates the necessary proof obligations. Note that this code is included in a non-deterministic

1: **if** (*) **then**
2: Let φ_1, φ_2 be Viper states s.t. $\varphi_1 \preceq \varphi_2 \preceq \mathit{currentViperState}$
3: $(\mathit{exist}, \mathit{currentViperState}) \leftarrow (\top, \varphi_1)$
4: $\mathit{guardExecs}(s, \mathit{exist})$
5: $\varphi_1' \leftarrow \mathit{currentViperState}$
6: $\mathit{currentViperState} \leftarrow \varphi_2$
7: s
8: **assert** $\mathit{exist} \land \varphi_1' \preceq \mathit{currentViperState}$
9: **assume** *false*
10: **end if**

Fig. 8. Proof obligation, expressed via self-composition, to automatically check if a deterministic statement s is structurally mono. We use pseudocode here, but the check can be expressed directly in Viper's verification condition generator based on Boogie.

branch (line 1), which is killed after the check (line 9). This allows us to include the check in the encoding of the inlined program without affecting the rest of its verification.

According to Def. 5.1, structural mono is defined relative to an upper bound T, which in the structural condition is instantiated with the set of states reachable before the statement s in the inlined program. These states are implicitly represented by the current verification state of the Viper program before statement s; in Fig. 8, we refer to this state as *currentViperState*; like all our states, it consists of a heap, a permission mask, and a store.

To prove structural mono for all states φ_1, φ_2, we choose (in line 2) fresh states non-deterministically and constrain them as prescribed by Def. 5.1 (where the current verification state represents $\varphi_3 \in T$). Viper does not have a built-in order on states but, as we explained above, we can express this easily via quantification over the contents of the heap and the mask.

Structural mono may assume that s verifies successfully in state φ_1 ($\mathit{ver}_{\mathcal{A}}(\varphi_1, s)$ in Def. 5.1). We achieve this by setting the current state to φ_1 (line 3), execute s (line 4), and record the final state as φ_1' (line 5). However, since the successful verification of s in φ_1 is an *assumption* of structural mono, we need to catch situations where this verification fails. In that case, we would incorrectly report an error even though structural mono is not violated. We also need to detect if the execution of s is infeasible because this would make the remaining proof obligations hold vacuously, even though structural mono is actually violated in this case. We solve both issues by executing a modified version of s, namely $s' \triangleq \mathit{guardExecs}(s, \mathit{exist})$, that avoids *infeasible* executions by accumulating assumptions using a fresh boolean variable *exist*, and by executing statements in s only if *exist* holds. s' avoids *failing* executions by turning assertions into assumptions. Hence, after the execution of s', if *exist* holds, then φ_1' corresponds to an output state of a verifying execution of s in φ_1, as required by Def. 5.1. We illustrate the transformation $\mathit{guardExecs}(\_, \mathit{exist})$ on the right of Fig. 7.

After this first (instrumented) execution of s in the state φ_1, we execute the (non-instrumented) statement s in the state φ_2 (lines 6 to 7). If no error is reported during this execution, we can guarantee that s is *safety* monotonic for the bound T. We check whether s is also *structural output monotonic*, which corresponds to the second conjunct on the implication's right-hand side of the structural mono definition (marked by "structural output mono." in Def. 5.1), as follows. Since we assume s to be deterministic (and, thus, $\mathit{det}(\varphi_1, \varphi_2, \varphi_2', s) = \mathit{sem}_{\mathcal{A}}(\varphi_1, s)$), Def. 5.1 requires $\mathit{sem}_{\mathcal{A}}(\varphi_1, s) \neq \emptyset$. In other words, it requires the first execution of s to reach a final state, that is, to be feasible, which we check by asserting that *exist* holds (line 8). Moreover, the final state of the second execution of s (φ_2' in Def. 5.1) must be larger than the final state of the first (φ_1' in Def. 5.1), which we assert as well. Note that the universal determinization condition must hold only for

$\varphi_2' \in sem_{\mathcal{A}}(\varphi_2, s)$. This is automatically the case in our proof obligation: if the execution of s on line 7 is infeasible, then the check on line 8 holds trivially.

In summary, the proof obligation from Fig. 7 reflects directly the definition of structural mono for deterministic statements s. Even though Def. 5.1 expresses a non-trivial hyperproperty that compares entire states, the resulting proof obligations can be proved automatically using standard verification tools. Next, we discuss how to check the property for non-deterministic statements.

Determinization. The determinization function $det(\varphi_1, \varphi_2, \varphi_2', s)$ used in Def. 5.1 yields the subset of final states $sem_{\mathcal{A}}(\varphi_1, s)$ of executions that make, wherever possible, the same non-deterministic choices as the execution that starts in φ_2 and ends in φ_2'. Here, the states correspond to the states from Def. 5.1; in particular, $det(\varphi_1, \varphi_2, \varphi_2', s)$ is meaningful when $\varphi_1 \preceq \varphi_2$. The corresponding proof obligation when s is deterministic (and thus $det(\varphi_1, \varphi_2, \varphi_2', s) = sem_{\mathcal{A}}(\varphi_1, s)$) is shown in Fig. 8.

However, for non-deterministic statements s, using the entire set of final states $sem_{\mathcal{A}}(\varphi_1, s)$ would lead to an overly strong definition of structural mono: Def. 5.1 would compare final states φ_1' and φ_2' obtained by making different non-deterministic choices and, for that reason, fail to satisfy $\varphi_1' \preceq \varphi_2'$. To avoid this problem, determinization aligns the two executions of s, to obtain a more relevant (and smaller) subset $det(\varphi_1, \varphi_2, \varphi_2', s)$ of $sem_{\mathcal{A}}(\varphi_1, s)$ (recall that the universal determinization condition is $\forall \varphi_1' \in det(\varphi_1, \varphi_2, \varphi_2', s).\, \varphi_1' \preceq \varphi_2'$). Instead of comparing every pair of executions (E_1, E_2) of statement s starting in φ_1 and φ_2 and ending in φ_1' and φ_2', respectively, determinization compares only those executions that resolve non-determinism similarly. To achieve this, we record all non-deterministic choices (such as the initial values of variables and newly-owned heap locations, or the existential fractional ownerships that have been chosen) made during the execution E_1 (line 4 in Fig. 8), and then ignore the pair (E_1, E_2) if E_2 (line 7) resolves non-determinism in a different way, provided that there is (at least) one pair with E_2 that is not ignored. The latter proviso ensures that the set of executions E_1 that are compared with E_2 is non-empty, i.e., that $det(\varphi_1, \varphi_2, \varphi_2', s) \neq \emptyset$, as required by the structural mono property (Def. 5.1). See App. G of the TR for more details.

6 EVALUATION

In this section, we evaluate four important aspects of our technique. We demonstrate that (1) features that may cause inlining to be non-preserving are widely used, (2) non-preserving inlining actually occurs in practice, (3) our structural condition is sufficiently precise, that is, it captures most examples that violate the syntactic condition but can be inlined in a verification-preserving manner, and (4) our implementation of verification-preserving inlining in Viper effectively detects bugs.

Our evaluation considers the test suites from VeriFast (1002 files), GRASShopper (314 files), Nagini (232 files), and RSL-Viper (14 files); the latter two encode verification problems into Viper. All four verifiers are interesting subjects for our evaluation because they use automation techniques and generate proof obligations for features that potentially lead to non-preserving inlining and that violate our syntactic condition. Examples from other tools, such as Prusti [Astrauskas et al. 2019], can always be validated by our syntactic condition, which demonstrates the usefulness of this check, but makes those tools less relevant for our evaluation.

We have implemented verification-preserving inlining for loops and method calls in Viper, taking partial annotations into account. Our implementation automatically checks the structural condition using the technique described in Sect. 5 (and omits it when the syntactic condition holds). Both inlining and checking the structural condition are performed by extending Viper's verification condition generator, which translates Viper programs to Boogie [Leino 2008]. The tool and examples are part of our publicly available artifact [Dardinier et al. 2023].

In the following, we refer to files (resp. features) that violate the syntactic condition as *non-trivial files* (resp. *non-trivial features*).

Table 1. Results of our syntactic checks and subsequent manual analysis for files from the test suites of RSL-Viper (R), Nagini (N), GRASShopper (G), and VeriFast (VF). The results show that 67% of the tests include features that our syntactic checks do not capture, that inlining may be non-preserving for the methods in 31% of the manually analyzed test cases, and that our structural condition is sufficiently precise to validate 94% of the test cases that are always-preserving.

	R	N	G	VF	Total
All files	14	232	314	1002	1562
└ Satisfy syntactic condition	-	-	203	306	509
└ Violate syntactic condition	14	232	111	696	1053
└ Manually analyzed	12	20	20	20	72
└ Not always-preserving	8	4	2	8	22
└ Always-preserving	4	16	18	12	50
└ Validated by semantic condition	4	15	18	12	49
└ Validated by structural condition	2	15	18	12	47
└ Lines of code (mean / median)	85 / 104	73 / 47.5	124 / 57	160 / 67	139 / 61

6.1 The Syntactic Condition is Often Violated

The first three rows in Tab. 1 show that non-trivial features appear often in the analyzed test suites: Out of 1562 files in total, 1053 (67%) violate the syntactic condition (the numbers were obtained via automatic detection of non-trivial patterns). This shows not only that most files contain features that might make inlining non-preserving, but also that our structural condition, which is more fine-grained than the syntactic one, is indeed necessary to determine whether inlining is verification-preserving. Proof rule selection strategies that depend on the owned resources are applied in 5 RSL-Viper files (out of 14, 36%) and 64 Nagini files (out of 232, 28%). Moreover, we found 111 non-trivial files in GRASShopper (out of 314, 35%) and 696 in VeriFast (out of 1002, 69%), mostly because of imprecise assertions that appear in predicate bodies, specifications, and ghost code. Apart from these two scenarios that we have discussed throughout the paper, there is a third scenario that violates the syntactic condition. 11 GRASShopper files (out of 314, 4%) and all Nagini files contain assertions that specify exact bounds on the resources owned. GRASShopper's `assert R` statement succeeds iff the method owns *exactly* the resources specified by R, reflecting GRASShopper's underlying classical SL. Nagini asserts at the end of each method that there are no remaining obligation resources to release a lock, which would get leaked when the method terminates. Since inlining affects the resources owned, these instances can also lead to non-preserving inlining.

6.2 Non-Preserving Inlining Occurs in Practice

Examples that violate our syntactic condition do not necessarily make inlining non-preserving. To assess whether inlining is actually non-preserving for non-trivial examples, we further analyzed non-trivial files from the four verifiers. Since most methods in the test suites have no (or very few) clients that invoke them, it would be insufficient to check whether non-preserving inlining occurs only for the existing clients as the initial statement, or for a fixed selection of inlining bounds. Instead, we (manually) analyze the methods for any possible client code (with minor restrictions) and any inlining bound. In particular, we classify a file as *always-preserving* if inlining every method in every caller context that satisfies the syntactic condition (that is, does not itself make inlining non-preserving)[12] is verification-preserving for every inlining bound. In RSL-Viper and Nagini, we analyzed the methods in the corresponding Viper encoding. For 114 files in Nagini,

[12]For Nagini, we used further restrictions on the clients to avoid that systematic leak checks prevent *all* examples from being always-preserving, which would not faithfully reflect typical clients.

Table 2. Non-trivial examples from the test suites of Nagini (N), RSL-Viper (R), VeriFast (VF), GRASShopper (G). VeriFast and GRASShopper examples were translated manually to Viper. We show the lines of code and lines of annotations needed for successful modular verification. The next three columns indicate whether inlining is verification-preserving (Inl.P.), the semantic condition holds (SC), and the structural condition holds (Str.C.). The verification time with inlining (T) is the average of 5 runs on a Lenovo T480 with 32 GB, i7-8550U 1.8 GhZ, Windows 10. The last two columns show the number of seeded errors (#Err.) to be found with inlining and the number of spurious errors reported when verifying modularly, but without annotations (#Sp.Err.). If more than one initial statement was considered in which calls were inlined, then the number of spurious errors is given by the average of spurious errors reported for each of the initial statements.

Name	LOC	Ann.	Inl.P.	SC	Str.C.	T [sec]	#Err.	#Sp.Err.
N_1: iap_bst	122	22	✓	✓	✓	19.2	2	10.3
N_2: parkinson_recell	37	9	✓	✓	✓	11.9	3	5.4
N_3: watchdog	52	9	✓	✓	✓	10.8	1	3
N_4: loops_and_release	20	n/a	✗	✗	✗	9.0	n/a	n/a
R_1: rust_arc [Doko and Vafeiadis 2017]	26	6	✓	✓	✓	3.4	7	1.6
R_2: lock_no_spin	17	2	✓	✓	✗	42	0	1
R_3: msg_pass_split_1	10	3	✓	✓	✓	2.7	1	5
R_4: msg_pass_split_2	10	n/a	✗	✗	✗	5.8	n/a	n/a
VF_1: account	43	8	✓	✓	✓	2.2	2	3.7
VF_2: lcp [Jacobs et al. 2014]	54	n/a	✗	✗	✗	7.3	n/a	n/a
VF_3: iterator	49	8	✓	✓	✓	1.6	2	4.5
VF_4: stack	50	6	✓	✓	✓	1.9	3	2.5
G_1: bstree	100	n/a	✗	✗	✗	13.5	n/a	n/a
G_2: nodes	54	29	✓	✓	✓	1.4	3	7.8

we could automatically deduce that they were always-preserving using extended verifier-specific syntactic checks. For the remaining files, manual inspection was required. For all verifiers except GRASShopper, some of these files were too complex for manual inspection, so we automatically discarded all those files with too many complicated features. This still left us with a large and diverse set of examples (12 for RSL-Viper, 79 for Nagini, 111 for GRASShopper, 271 for VeriFast). From these examples, we picked 20 examples randomly for each verifier (except RSL-Viper, where we took all). The results of this manual analysis are presented in Tab. 1. We took existing annotations into account. Not doing so would have resulted in a different classification for only 2 examples in VeriFast, which would have been classified as always-preserving instead.

Overall, out of the 72 non-trivial files that we analyzed manually, 22 files (31%) are not always-preserving. This shows that inlining is non-preserving in a non-negligible number of cases, and that inlining is verification-preserving in the majority of cases even when the syntactic condition is violated; thus, our more-precise structural condition is needed to validate those (see Sect. 6.3).

Our manual inspection revealed that in VeriFast, non-preserving inlining is often due to imprecise assertions. In RSL-Viper, the non-preserving pattern from Fig. 2 occurs in 5 examples. In GRASShopper and Nagini, the main source of non-preserving inlining are assertions on exact bounds of resources, which are non-preserving in calling contexts that provide more resources.

6.3 Our Conditions are Effective and Precise

Our semantic and structural conditions are sufficient for inlining to be verification-preserving, but not necessary. To evaluate the precision of the conditions, we further examined examples for which inlining is always-preserving. For the 114 automatically handled examples in Nagini, we could also automatically deduce that both the semantic and structural conditions hold. For the 72 manually handled examples, we also evaluated the conditions manually and will discuss the results (shown in Tab. 1) next. We also assessed the usefulness of the bounded relaxation of our condition.

Our semantic condition is precise. As shown by the results of our manual analysis (Tab. 1), our semantic condition captures almost all (49 out of 50, 98%) non-trivial files classified as always-preserving. Besides the always-preserving file (in Nagini) not captured by our condition, there are methods in the non-preserving files for which inlining is non-preserving in *some* caller contexts, but verification-preserving in others. Our manual inspection revealed some verification-preserving caller contexts that our semantic condition cannot validate. We found such patterns in RSL-Viper (see Fig. 2), VeriFast, and Nagini, but not in GRASShopper. Dealing with these patterns requires a non-compositional approach taking the entire program into account, which is practically infeasible.

Our structural condition is effective. Our structural condition is stronger than the semantic condition. Tab. 1 shows that this over-approximation is very precise in practice: the structural condition validates 96% of the always-preserving files that satisfy the semantic condition.

The bounded relaxation is required. We argued in Sect. 3.4 that the bounded relaxation of our condition is needed to validate common patterns. Our manual inspection confirmed this claim. For example, Nagini uses two patterns (occurring in 62 and all 232 test cases, resp.) that can be validated only with the bounded conditions. In GRASShopper and VeriFast, imprecise assertions (occurring in 107 and all 697 test cases, resp.) are often unambiguous in the context they are inlined in, and thus satisfy our structural condition only because of the bounded relaxation.

6.4 Verification-preserving Inlining Effectively Detects Bugs

The structural condition can be checked automatically. The previous subsection showed that our structural condition is sufficiently precise. To evaluate whether it can be checked automatically, we manually selected, out of the 1053 non-trivial files, a diverse set of examples (shown in Tab. 2) that reflect the non-trivial patterns occurring in the different verifier test suites and that could be translated to Viper. Our tool correctly reports whether the structural condition holds in all cases irrespective of whether existing annotations are taken into account.

Inlining detects errors effectively. To evaluate how effective inlining is in finding true errors without annotations, we consider all examples in Tab. 2 for which inlining is verification-preserving, and some examples taken from the Viper test suite, most of which satisfy the syntactic condition (Tab. 3 in App. H of the TR). We seeded errors by making simple changes in the implementations or writing clients that use methods incorrectly. For several examples, we considered more than one initial statement in which calls were inlined and loops were unrolled. Our tool was able to report every true error for some sufficiently large inlining bound (bounds between 1 and 4 were sufficient for all examples except for N_4 and G_2 in Tab. 2, where bounds 10 and 11 were required since both examples contain a loop that iterates 10 times). Our tool never reported a spurious error, which was expected since our structural condition is sufficient to ensure that inlining is verification-preserving.

Inlining reduces annotation overhead. Verifying the same examples modularly without providing any annotation results in at least one spurious error each, and 3 on average. To assess the amount of annotations saved by using inlining instead of modular verification, we considered annotations for all examples to successfully verify without inlining (that is, to show they are memory safe and satisfy all provided assertions). This required 256 lines of annotation in the programs for 1152 lines of code, which inlining does not require. This result shows that inlining is useful to find true errors with low annotation overhead and to gain confidence that an implementation is correct.

6.5 Threats to Validity

We identified the following threats to the validity of our evaluation.

Dataset. Our examples, which we took from the test suites of VeriFast, GRASShopper, Nagini, and RSL-Viper, might not be representative of realistic code. We believe this threat to be minor since (1) these test suites contain various practically-relevant verification problems and (2) contain non-trivial specifications and exercise features that show up in real-world programs.

As we discussed in Sect. 6.2, we discarded some files automatically before choosing files randomly for the manual analysis presented in Sect. 6.2 and 6.3. We are convinced that discarding these files does not compromise the validity of our manual analysis (and accompanying automatic analysis for Nagini) for the following reasons. In VeriFast, we discarded files that were too large for manual inspection. Features that violate the syntactic condition in VeriFast often appear within small recurring patterns that also show up in the considered smaller files. We are thus confident that our results for "always-preserving" files would be similar for the discarded files. Since the discarded files are larger, we likely would get more files that exhibit non-preserving inlining, which would still support our conclusion that non-preserving inlining occurs. For Nagini and RSL-Viper, we discarded files that contain features that were too complex to analyze manually. However, in both cases, we still consider the vast majority of the test cases (83% and 86%, respectively).

Manual analysis. We may have made mistakes in our manual analysis presented in Sect. 6.2 and 6.3. We mitigated this risk as follows. When a file is not always-preserving, we wrote a simple client satisfying the syntactic condition that invokes a method in the file to confirm this observation. That is, we (1) wrote an annotation for which the program (consisting of the client as the initial statement) verifies modularly and (2) identified an inlining bound for which the inlined program does not verify. To check (2), we used our inlining tool for Viper-based verifiers; for GRASShopper and VeriFast, we inlined calls and unrolled loops manually, and then used the corresponding verifier. When inlining is always-preserving for a file, we sketched informal proofs for every method in the file. We did so by considering each statement of a method that does not satisfy the syntactic condition, and reasoned why (given the rest of the method) it cannot lead to non-preserving inlining. We sketched similar proofs in cases where the semantic condition and structural condition hold.

Error seeding. We seeded the errors in Tab. 2 and Tab. 3 in App. H of the TR mostly ourselves. We tried to mitigate a potential bias by seeding different kinds of errors (e.g., asserting incorrect properties in clients, using libraries incorrectly, erroneously adjusting library implementations).

7 RELATED WORK

Corral [Lal et al. 2012] detects bugs in C programs by translating them to Boogie [Leino 2008]. The Boogie verifier is used to check correctness of the inlined Boogie program. Inlining is trivially verification-preserving for Corral. Additionally, Corral applies various techniques to improve efficiency such as approximating method calls with inferred *method call summaries* without inlining them. Lourenço et al. [2019] consider bounded verification using inlining in the context of a standard verification language without any resources. They directly connect correctness of the original program (instead of verification w.r.t. a verifier semantics) to verification of the inlined program, which is not possible in our setting due to proof search algorithms.

One of our motivations as mentioned in Sect. 1 is to use bounded verification as a stepping stone for modular verification. This is also the case for Beckert et al. [2020], who define a translation from a Java program with annotations expressed in the Java Modeling Language [Leavens et al. 2006] to a Java program that is accepted by the JBMC bounded model checker [Cordeiro et al. 2018]. Bounded model checkers (BMC) such as JBMC and CBMC [Clarke et al. 2004] perform bounded verification (via inlining) and detect errors effectively but do not support annotations such as method contracts and frame conditions, and generally support less expressive assertions than deductive verifiers.

Instead of using an off-the-shelf BMC, we inline the program and then use already-existing automatic deductive SL verifiers, which have a mature automation infrastructure for SL assertions. This allows us to directly support inlining partial annotations or calls to library methods without additional work on SL assertion support in BMC. Moreover, performing both kinds of verification within the same tool ensures that no verification errors are caused by switching from one tool to the other, for instance, due to small differences in the verifier semantics. Nevertheless, it would be interesting to explore a BMC technique that supports the handling of SL assertion logics such as those from GRASShopper, Viper and VeriFast (potentially building on SL runtime checking [Agten et al. 2015; Nguyen et al. 2008] or SL model checking [Brotherston et al. 2016]).

While safety monotonicity and the framing property have been studied in the context of SL [Calcagno et al. 2007; Yang and O'Hearn 2002], the relaxed conditions we use (where the states these conditions quantify over are bounded using the inlined program's execution) have, to the best of our knowledge, not been explored before. As shown in Sect. 6, these relaxations are essential to capture common idioms. Moreover, our semantic condition imposes monotonicity and framing constraints on *different* parts of the program (based on the relationship between modular and inlined verification), which is crucial to capture common patterns that satisfy these properties but contain statements that do *not*, as illustrated in Sect. 3. Finally, we propose a novel output monotonicity property, which, to the best of our knowledge, has not been used in the context of SL.

Several automatic SL verifiers use incomplete heuristics in their proof search strategies that may lead to non-preserving inlining, such as Caper [Dinsdale-Young et al. 2017], GRASShopper [Piskac et al. 2014], Nagini [Eilers and Müller 2018], RefinedC [Sammler et al. 2021], RSL-Viper [Summers and Müller 2018], Steel [Fromherz et al. 2021], VerCors [Blom et al. 2017], VeriFast [Jacobs et al. 2011], and Viper [Müller et al. 2016]. RefinedC uses incomplete rules and Steel uses incomplete heuristics to instantiate existentially-quantified variables; both may lead to non-preserving inlining (Fig. 13 and 14 in App. I of the TR). Caper uses backtracking when resolving non-deterministic choices to make the proof search more complete. However, for the *region creation* proof rule, which can (but need not) be applied at various points, Caper cannot explore all options. Instead, it uses incomplete heuristics that can lead to non-preserving inlining (Fig. 12 in App. I of the TR). While our framework can be applied to GRASShopper, VeriFast, and verifiers based on Viper, it cannot be applied to Caper, because backtracking does not fit into our formal model.

8 CONCLUSION

We demonstrated that inlining may introduce false positives when using automatic SL verifiers. Their automation techniques are sensitive to changes in ownership, which occur inevitably during inlining. We identified a novel compositional semantic condition and proved that it is sufficient to ensure verification-preserving inlining. Since this condition is difficult to check, we developed two approximations that can be checked syntactically and with a standard program verifier, respectively. Our evaluation shows that these conditions are necessary and capture most use cases.

Our work paves the way to bounded verification within automatic SL verifiers without the risk of false positives. We believe that the foundations presented in this paper can also be used for other applications, such as the caching of verification results in automatic SL verifiers. Existing caching approaches [Leino and Wüstholz 2015] do not re-verify code after a call if the postcondition of the callee is strengthened. Such techniques may be unsound when applied to automatic SL verifiers if the statement after the call is not safety monotonic. One direction for future work is to devise a sound caching approach for automatic SL verifiers using our techniques.

ACKNOWLEDGMENTS

We thank the following people for their help: Vytautas Astrauskas and Marco Eilers (Nagini), Felix Wolf (Caper), Michael Sammler (RefinedC), Denis Merigoux (Steel), Malte Schwerhoff (VeriFast), and Christoph Matheja (giving feedback on previous drafts). This work was partially funded by the Swiss National Science Foundation (SNSF) under Grant No. 197065.

DATA AVAILABILITY STATEMENT

Our publicly-available artifact [Dardinier et al. 2023] contains:

(1) Isabelle/HOL proofs of the technical results from Sect. 4 and 5.
(2) An analysis of the test suites of GRASShopper, Nagini, RSL-Viper, and VeriFast, corresponding to the results shown in Tab. 1.
(3) The inlining tool for Viper, described in Sect. 6, which inlines calls and unrolls loops, while also checking the structural condition.
(4) A test framework that runs the inlining tool on the examples from Tab. 2 (and Tab. 3 from the TR [Dardinier et al. 2022b]).

REFERENCES

Pieter Agten, Bart Jacobs, and Frank Piessens. 2015. Sound Modular Verification of C Code Executing in an Unverified Context. In *Principles of Programming Languages (POPL)*, Sriram K. Rajamani and David Walker (Eds.). https://doi.org/10.1145/2676726.2676972

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

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure Information Flow by Self-Composition. *Mathematical Structures in Computer Science (MSCS)* 21, 6 (2011), 1207–1252. https://doi.org/10.1017/S0960129511000193

Bernhard Beckert, Michael Kirsten, Jonas Klamroth, and Mattias Ulbrich. 2020. Modular Verification of JML Contracts Using Bounded Model Checking. In *International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 12476. 60–80. https://doi.org/10.1007/978-3-030-61362-4_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*, Nadia Polikarpova and Steve Schneider (Eds.). Springer International Publishing, Cham, 102–110. https://doi.org/10.1007/978-3-319-66845-1_7

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

James Brotherston, Nikos Gorogiannis, Max I. Kanovich, and Reuben Rowe. 2016. Model Checking for Symbolic-Heap Separation Logic with Inductive Predicates. In *Principles of Programming Languages, POPL*, Rastislav Bodík and Rupak Majumdar (Eds.). 84–96. https://doi.org/10.1145/2837614.2837621

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *Logic in Computer Science (LICS)*. 366–375. https://doi.org/10.1109/LICS.2007.30

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Kurt Jensen and Andreas Podelski (Eds.). 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*. 51–65. https://doi.org/10.1109/CSF.2008.7

Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *Computer Aided Verification (CAV)*, Hana Chockler and Georg Weissenbacher (Eds.). https://doi.org/10.1007/978-3-319-96145-3_10

Thibault Dardinier, Peter Müller, and Alexander J. Summers. 2022a. Fractional Resources in Unbounded Separation Logic. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 163 (2022), 27 pages. https://doi.org/10.1145/3563326

Thibault Dardinier, Gaurav Parthasarathy, and Peter Müller. 2022b. Verification-Preserving Inlining in Automatic Separation Logic Verifiers (extended version). https://doi.org/10.48550/ARXIV.2208.10456

Thibault Dardinier, Gaurav Parthasarathy, and Peter Müller. 2023. *Verification-Preserving Inlining in Automatic Separation Logic Verifiers – Artifact.* https://doi.org/10.5281/zenodo.7711788

Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. 2022c. Sound Automation of Magic Wands. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 130–151. https://doi.org/10.1007/978-3-031-13188-2_7

Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 2017. Caper - Automatic Verification for Fine-Grained Concurrency. In *European Symposium on Programming (ESOP)*, Hongseok Yang (Ed.). https://doi.org/10.1007/978-3-662-54434-1_16

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Asian Symposium on Programming Languages and Systems (APLAS)*, Zhenjiang Hu (Ed.). 161–177. https://doi.org/10.1007/978-3-642-10672-9_13

Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *European Symposium on Programming (ESOP)*, Hongseok Yang (Ed.). 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

Marco Eilers and Peter Müller. 2018. Nagini: A Static Verifier for Python. In *Computer Aided Verification (CAV)*, Hana Chockler and Georg Weissenbacher (Eds.). 596–603. https://doi.org/10.1007/978-3-319-96145-3_33

Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473590

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (NFM)*. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Bart Jacobs, Jan Smans, and Frank Piessens. 2014. Solving the VerifyThis 2012 Challenges with VeriFast. *International Journal on Software Tools for Technology Transfer (STTT)* 17 (03 2014). https://doi.org/10.1007/s10009-014-0310-9

Ralf Jung, Robbert Krebbers, Jacques Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming (JFP)* (2018). https://doi.org/10.1017/S0956796818000151

Akash Lal and Shaz Qadeer. 2014. Powering the Static Driver Verifier Using Corral. In *International Symposium on Foundations of Software Engineering (FSE)*. 202–212. https://doi.org/10.1145/2635868.2635894

Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. 2012. A Solver for Reachability Modulo Theories. In *Computer Aided Verification (CAV)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. 427–443. https://doi.org/10.1007/978-3-642-31424-7_32

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Software Engineering Notes* 31, 3 (may 2006), 1–38. https://doi.org/10.1145/1127878.1127884

K. Rustan M. Leino. 2008. This is Boogie 2. (2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

K. Rustan M. Leino and Valentin Wüstholz. 2015. Fine-Grained Caching of Verification Results. In *Computer Aided Verification (CAV)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 380–397. https://doi.org/10.1007/978-3-319-21690-4_22

Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto. 2019. A Generalized Program Verification Workflow Based on Loop Elimination and SA Form. In *International Conference on Formal Methods in Software Engineering (FormaliSE)*. 75–84. https://doi.org/10.1109/FormaliSE.2019.00017

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*, B. Jobstmann and K. R. M. Leino (Eds.), Vol. 9583. Springer-Verlag, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. 2008. Runtime Checking for Separation Logic. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck (Eds.). https://doi.org/10.1007/978-3-540-78163-9_19

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

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Erika Ábrahám and Klaus Havelund (Eds.). 124–139. https://doi.org/10.1007/978-3-642-54862-8_9

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. *Logic in Computer Science (LICS)*, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Programming Language Design and Implementation (PLDI)*. 158–174. https://doi.org/10.1145/3453483.3454036

Malte Schwerhoff and Alexander J. Summers. 2015. Lightweight Support for Magic Wands in an Automatic Verifier. In *European Conference on Object-Oriented Programming (ECOOP) (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. 614–638. https://doi.org/10.4230/LIPIcs.ECOOP.2015.614

Alexander J. Summers and Peter Müller. 2018. Automating Deductive Verification for Weak-Memory Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, D. Beyer and M. Huisman (Eds.). Springer-Verlag, 190–209. https://doi.org/10.1007/978-3-319-89960-2_11

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Principles of Programming Languages (POPL)*. 256–270. https://doi.org/10.1145/2837614.2837655

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *Object Oriented Programming Systems Languages & Applications (OOPSLA)*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 867–884. https://doi.org/10.1145/2544173.2509532

Hongseok Yang and Peter O'Hearn. 2002. A Semantic Basis for Local Reasoning. In *Foundations of Software Science and Computation Structures (FoSSaCS)*. 402–416. https://doi.org/10.1007/3-540-45931-6_28

Received 2022-10-28; accepted 2023-02-25