Inductive Predicate Synthesis Modulo Programs (Extended)

Scott Wesley
Dalhousie University, Halifax, Canada

Maria Christakis
TU Wien, Vienna, Austria

Jorge A. Navas
Certora, Seattle, Washington, USA

Richard Trefler
University of Waterloo, Waterloo, Canada

Valentin Wüstholz
ConsenSys, Vienna, Austria

Arie Gurfinkel
University of Waterloo, Waterloo, Canada

Abstract

A growing trend in program analysis is to encode verification conditions within the language of the input program. This simplifies the design of analysis tools by utilizing off-the-shelf verifiers, but makes communication with the underlying solver more challenging. Essentially, the analysis tools operates at the level of input programs, whereas the solver operates at the level of problem encodings. To bridge this gap, the verifier must pass along proof-rules from the analysis tool to the solver. For example, an analysis tool for concurrent programs built on an inductive program verifier might need to declare Owicki-Gries style proof-rules for the underlying solver. Each such proof-rule further specifies how a program should be verified, meaning that the problem of passing proof-rules is a form of invariant synthesis.

Similarly, many program analysis tasks reduce to the synthesis of pure, loop-free Boolean functions (i.e., *predicates*), relative to a program. From this observation, we propose Inductive Predicate Synthesis Modulo Programs (IPS-MP) which extends high-level languages with minimal synthesis features to guide analysis. In IPS-MP, unknown predicates appear under assume and assert statements, acting as specifications modulo the program semantics. Existing synthesis solvers are inefficient at IPS-MP as they target more general problems. In this paper, we show that IPS-MP admits an *efficient* solution in the Boolean case, despite being generally undecidable. Moreover, we show that IPS-MP reduces to the satisfiability of constrained Horn clauses, which is less general than existing synthesis problems, yet expressive enough to encode verification tasks. We provide reductions from challenging verification tasks—such as parameterized model checking—to IPS-MP. We realize these reductions with an efficient IPS-MP-solver based on SeaHorn, and describe a real-world application to smart-contract verification.

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases Software Verification, Invariant Synthesis, Model-Checking

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.26

Funding *Maria Christakis*: supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22007].
Richard Trefler: supported, in part, by a Discovery Grant (Individual) from the Natural Sciences and Engineering Research Council of Canada.
Arie Gurfinkel: supported, in part, by a Discovery Grant (Individual) from the Natural Sciences and Engineering Research Council of Canada.

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 26; pp. 26:1–26:37
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

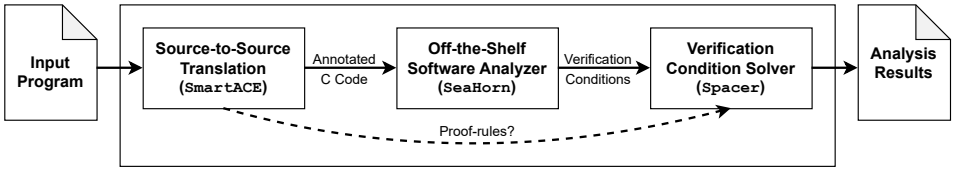

Figure 1 The architecture of an analysis framework built atop an off-the-shelf software verifier. Examples are given with respect to SmartACE.

1 Introduction

In recent years, many tools have emerged to verify C programs by leveraging the Clang/LLVM compiler infrastructure (e.g., [9, 61, 57, 55, 32]). These tools take as input C programs annotated with assumptions and assertions, and decide whether an assertion can be violated given that all assumptions are satisfied. One such tool is SeaHorn [32], which employs techniques from software model checking [43], abstract interpretation [33], and memory analysis [44] to enable efficient verification. Due to these features, many tool designers have started using annotated C code as an intermediate language to dispatch program analysis problems to SeaHorn (e.g., [56, 41, 4, 15, 18, 67]). In this setting, programs with specifications are transformed into C programs with assumptions and assertions, and then these C programs are analyzed using SeaHorn. The results obtained from SeaHorn are examined to draw conclusions about the input programs.

However, the flexibility afforded by C code as an intermediate language makes communication with the underlying verification algorithm more challenging. When SeaHorn is given a program to verify, it automatically applies various builtin proof-rules, such as induction for loops [3] and function summarization [43]. A tool designer has no control over how these rules are employed, nor is the developer able to introduce new proof-rules to SeaHorn. The goal of this paper is to extend SeaHorn with the language features required to communicate new declarative proof-rules to the underlying verification algorithm.

To illustrate this challenge, we consider SmartACE [67], a tool that uses SeaHorn for modular Solidity smart-contract verification. In SmartACE, each smart-contract is modeled by a non-terminating loop that executes a sequence of transactions[1]. For SmartACE to verify a smart-contract, it first requires an inductive invariant for the non-terminating loop, and a compositional invariant for each map[2] in the program. The discovery of an inductive invariant is automated by SeaHorn's invariant inference capabilities. However, SeaHorn is unaware of the modular proof-rules used by SmartACE, and therefore, the end-user must provide the compositional invariants manually. The authors of SmartACE hypothesized [67] that if each proof-rule could be declared to SeaHorn, then SeaHorn could instruct the underlying verification algorithm to infer all invariants automatically. Inspired by this hypothesis, we first implemented compositional invariant synthesis in SeaHorn, and then discovered that our solution generalized to many program verification problems. Consequently, our solution forms a general-purpose framework well-suited to compositional invariant synthesis.

To illustrate this more general problem, consider a tool designer who wishes to use an off-the-shelf software verifier (e.g., SeaHorn) as the back-end to a new analysis framework

[1] Transactions in Solidity/Ethereum can be thought of as sequences of method invocations.
[2] In Solidity, maps are often used to store data for individual smart-contract users.

(e.g., SmartACE). Recall that many off-the-shelf verifiers rely on specialized solvers to discharge verification conditions, including solvers for Satisfiability Modulo Theories [12], Constrained Horn Clauses (CHCs) [38], or intermediate verification languages (e.g., [10, 26]). As depicted in Figure 1, an analysis framework built atop an off-the-shelf verifier takes as input a program with specifications, translates this program into the language of the verifier, and then uses the verifier to generate verification conditions for its specialized solver. Since software verification is undecidable in general, it is often necessary for the tool designer to declare additional proof-rules for the solver. Example proof-rules include introducing predicate abstractions, suggesting modular abstractions for an array, and proposing modular decompositions for a parameterized system. However, it is challenging for the tool designer to communicate proof-rules to the solver—the former operates at the level of the input program, while the latter operates at the level of verification conditions. If a tool designer does attempt to encode proof-rules at the level of the input program, then these proof-rules are typically eliminated by optimizations from the verifier[3], long before verification conditions are ever produced. That is, there is an impedance mismatch!

To bridge this gap, the verifier must pass proof-rules from the tool designer to the solver. Each proof-rule is associated with a set of invariants that the solver must find in order to prove the program correct. In other words, the invariants are declared by the proof-rules. Since these invariants span many classes (e.g., inductive, compositional, and object invariants), it is often the case that specialized invariant inference techniques cannot solve this problem. Instead, one must note that each proof-rule refines the invariants which the solver must synthesize. Consequently, one solution to the aforementioned impedance mismatch is to use synthesis techniques (e.g., [7, 25, 71, 60]). In particular, using synthesis allows the tool designer to declare proof-rules by specifying what invariants are to be synthesized at the level of the input program. This flexibility, however, comes at a price. General synthesis is significantly more expensive than verification [65]!

Our key contribution is a definition of *a new form of synthesis*, called *Inductive Predicate Synthesis Modulo Programs* (IPS-MP), that bridges the gap between flexible verification and efficient synthesis. Our *theoretical results* are two-fold, we show that: (a) IPS-MP reduces to satisfiability of CHCs, hence establishing that IPS-MP is a specialization of general synthesis [62, 64, 5, 42]; (b) for the special case of Boolean programs, IPS-MP is decidable with the same complexity as verification. We conjecture that the latter extends to other decidable models of programs (e.g., timed automata). Our *practical result* is to reduce a wide range of common proof-rules to IPS-MP. We show how IPS-MP guides inference of inductive invariants, class invariants, array invariants, and even modular parameterized model checking. In other words, IPS-MP is well-suited to many areas of program analysis. As a real-world application, we show that IPS-MP enables the full automation of SmartACE.

Similar to existing synthesis frameworks, IPS-MP extends a programming language with unknowns. The language itself is unrestricted (i.e., it has loops, procedures, memory, etc.). However, the unknowns may only appear within `assume` and `assert` statements, denoting constraints on the strongest and weakest possible solutions, respectively. A solution to an IPS-MP problem is a mapping from each unknown to a Boolean predicate such that the resulting program is correct (i.e., satisfies all of its assertions). A high-level overview of IPS-MP is shown in Figure 2. Each problem instance consists of two components: (1) a program with its specification (described by assumptions and assertions), which contains calls to unknown predicates under assume and assert, and (2) the declarations of those

[3] For example, a pure function with annotations may be optimized away by the Clang compiler.

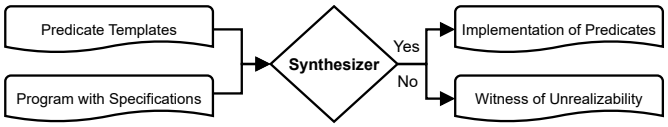

Figure 2 Overview of the IPS-MP problem.

predicates, which we refer to as *predicate templates*. Intuitively, a predicate template is a partial implementation with a number of unknown statements. A solution consists of a full implementation of each predicate, or a witness to unrealizability (i.e., a proof that a solution does not exist).

The reducibility of IPS-MP to CHC-solving motivates an efficient IPS-MP solver. We build on the SeaHorn framework (thus, our underlying language is the fragment of C supported by SeaHorn [32]), and integrate with two CHC solvers, namely Spacer [43], and Eldarica [35]. Our empirical results on verification problems from various domains show that: (1) IPS-MP is effective at specifying verification strategies, (2) our implementation combined with existing CHC-solvers is highly efficient for linear arithmetic invariants, and (3) existing reductions to either general synthesis or specification inference are infeasible. Our evaluation focuses on general synthesis, rather than invariant inference, since the invariants in our benchmarks span many classes. To contextualize these results, we briefly review the state-of-the-art in synthesis.

State-of-the-art in synthesis. The general synthesis problem is the task of generating a program that satisfies a given specification. There are many general synthesis frameworks, e.g., Sketch [62], Rosette [64], SyGuS [5], and SemGuS [42]. In Sketch and Rosette, users write programs with *holes*, representing unknowns. These holes are filled with predefined, loop-free expressions such that all program assertions are satisfied. SyGuS introduced a more language-agnostic approach to general synthesis. It generates loop-free *programs*, satisfying a given behavioral specification, from a potentially infinite language. Building on SyGuS, SemGuS allows users to define pluggable semantics, thereby enabling synthesis of programs with loops. A distinguishing characteristic along this line of work is an emphasis on software development. In contrast, IPS-MP targets software verification and proof synthesis, which are theoretically simpler problems.

Specification synthesis (e.g., [21, 2, 54]) is another line of work that addresses a more specialized synthesis problem targeting program analysis, rather than software development. In specification synthesis, a program may call functions with unknown implementations. The goal is to synthesize specifications (e.g., the weakest specification for an unknown library procedure) that ensure the correctness of the calling program. Typically, a specification synthesizer imposes extra requirements, such as non-vacuity [54], maximality [2], or reachability [21], to ensure that solutions are reasonable. In contrast, the invariants synthesized by IPS-MP have constraints on both the strongest and weakest possible solutions, avoiding the need for additional (and often costly) requirements.

Of particular interest are the similarities and differences between IPS-MP and syntax-guided synthesis. In IPS-MP, program holes are filled by expressions from an unbounded language. To make this problem tractable, IPS-MP restricts Sketch and Rosette by requiring that holes only appear in partial predicates. Formally, this means that IPS-MP solving is subsumed by non-linear constrained Horn clause solving. This restriction is crucial as it allows an IPS-MP solver to prove that a problem is unrealizable, unlike in Sketch or Rosette. Furthermore, IPS-MP differs from SyGuS and SemGuS in that the behavioral specification is given with respect to a given program (in other words, modulo a given program), rather

than through a separate logical specification. The program itself also places requirements on the holes, through assumptions and assertions, which is in contrast to specification synthesis.

In recent years, new extensions have been proposed to the Sketch framework. However, these extensions all generalize Sketch to more complex, and consequently less tractable, problems, whereas IPS-MP restricts Sketch to a more tractable problem which proves to be useful in the domain of program verification. To illustrate these gaps, we compare IPS-MP to PSKETCH [63], Synapse [16], Grisette [47], and MetaLift [13]. In the case of PSKETCH, both frameworks target the development of provably correct concurrent programs. However, PSKETCH focuses on inductive program verification in the presence of interleaving executions, whereas IPS-MP focuses on the verification of sequential code fragments via user-defined proof rules (e.g., the synthesis of compositional invariants in SmartACE). In the case of Synapse, both tools aim to extend program synthesis problems with hints provided by an end-user. However, the nature of these hints is very different. In IPS-MP, the user introduces entirely new proof-rules, for which an underlying solver oversees the search for a solution. In contrast, the hints provided by an end-user to Synapse assign costs to solutions, for which the underlying solver tries to optimize. These hints do not allow the end-user to propose new proof-rules, and are suited to synthesis optimization rather than program verification. In the case of Grisette, a framework was proposed to programmatically generate and solve sketches. However, Grisette is based around bounded model-checking, whereas the IPS-MP problem targets unbounded model-checking and is, therefore, incomparable. More closely related is MetaLift, which makes use of the fact that inductive program verification can be reduced to syntax-guided synthesis. However, this verification program is not exposed to end-users. In particular, the assume and assert statements are hidden from end-users, and the end-user has no way to propose new placements for them. We conclude that IPS-MP is a novel synthesis problem.

Constrained Horn clauses. A prominent approach to verification is reduction to the satisfiability of CHCs, otherwise known as *verifier synthesis* [30]. While verifier synthesis does enable the flexible design of software verifiers, it does not address the issue of communicating proof-rules to the underlying solver. In invariant synthesis, the proof-rules are either chosen once and for all [30], or are implicit in the solving algorithm (e.g., [45, 66]). While we show that IPS-MP reduces to CHC-solving, our focus is on communicating new proof-rules to the solver via synthesis. Other solutions to IPS-MP might emerge in the future.

Contributions. This paper makes the following contributions:

1. Sec. 4 presents the novel IPS-MP problem which has many applications to both program analysis and software verification;
2. Sec. 5 shows that even though IPS-MP is undecidable in general, there exists an efficient solution modulo Boolean programs;
3. Sec. 6 provides reductions from important verification problems to IPS-MP;
4. Sec. 7 presents a solver for IPS-MP within SeaHorn. We demonstrate the effectiveness of our implementation compared to state-of-the-art synthesis frameworks CVC4 [11] (a SyGuS synthesizer) and HornSpec [54] (a specification synthesizer). We conclude that IPS-MP fills a gap not met by other synthesis frameworks.

All omitted proofs are found in the appendix.

2 Overview

To illustrate the basics of IPS-MP, we start with an artificial example. For the moment, we focus on the language used in our presentation and the possible solutions to an IPS-MP

```
1  bool PRED_TEMPLATE Post(int x, int y) {
2    return synth(x, y); }
3  void main(int y) {
4    int x = 0;
5    assume(y > 0);
6    for (int i = 0; i < y; ++i) {
7      x+=1; }
8    assert(Post(x, y));
9    x = *; y = *; assume(Post(x, y));
10   assert(x == y); }
```

Figure 3 A simple example of the IPS-MP problem.

problem. Realistic applications of IPS-MP, highlighting its importance, are presented later in this section.

Our example, shown in Figure 3, consists of a single function `main` that provides the context for a synthesis problem. The function `main` is written in a typical imperative language, with loops and function calls. We extend the language with two verification statements, `assume` and `assert`, with their usual semantics. In our example, `y` is initially positive, due to `assume(y > 0)` on line 5, and the program is correct if `assert(x == y)` on line 10 holds for all executions. That is, lines 5 and 10 provide a program specification. The goal of this example is to synthesize a pure expression e such that the program is correct after substituting e for each call to `Post`. To indicate that a predicate is a target for synthesis, the language is extended by the predicate template annotation `PRED_TEMPLATE` (line 1). Each predicate template is a pure, loop-free function whose body either returns `true`, or returns via a call to the special predicate `synth`. Each call to `synth` indicates a *hole* in the predicate implementation, and must be determined by a synthesizer. Each return of `true` places an *explicit constraint* on when the implementation must be true. In our example, `Post` always returns via a call to `synth` (line 2). In the rest of the program, a call to a partial predicate can only appear as an argument to either `assume` or `assert`. As described below, verification calls place *implicit constraints* on when the implementation must be true. Multiple calls to the same predicate are allowed. In our example, `Post` is called once under `assert` (line 8), and once under `assume` (line 9).

A *solution* to an IPS-MP problem is a mapping from each partial predicate p to a pure Boolean expression e over the arguments of p, such that if every call to `synth` in p is replaced by e, then the main program satisfies all of its assertions. If such a solution does not exist, the output is a *witness to unrealizability*, which is a mapping from each partial predicate p to a pure Boolean expression e over the arguments of p, which is both necessitated by the assertions placed on the partial predicate, and sufficient to violate an assertion that is part of the specification. In our example, there are many possible solutions. The weakest and strongest solutions are $post_{weak}(x, y) = (x = y)$ and $post_{strong}(x, y) = (y > 0 \wedge x = y)$. Each solution defines a corresponding predicate `Post` such that all assertions in the `main` program are satisfied. Intuitively, each call to `Post` under `assume` provides an implicit constraint on the weakest possible synthesized solution. Likewise, each call to `Post` under `assert` provides an implicit constraint on the strongest possible synthesized solution. Following this intuition, the example shows an application of IPS-MP to find an intermediate post-condition, over two variables `x` and `y`, that is true after the loop and is strong enough to ensure an assertion. This means that solving IPS-MP requires, in general, inferring inductive invariants for loops and summaries for functions.

To illustrate the case when synthesis is not possible, consider removing line 7 from Figure 3. Since `x` is not incremented, it will never equal `y`. However, `Post` cannot be mapped to `false`, since this violates the assertion on line 8. If `Post` is not `false`, then the assertion on line 8 is reachable and will fail. Therefore, this IPS-MP problem is unrealizable. The witness to unrealizability is a mapping that sends `Post` to an expression over `x` and `y`, which is necessitated by the assertion on line 8 and violates the assertion on line 10. An example

witness is $synth_{witness}(x, y) = (x = 0 \wedge y = 1)$.

This section continues with three important applications of IPS-MP. Sec. 2.1 presents a methodology to reduce verification problems to IPS-MP. For readers new to verification as synthesis, the standard example of inductive loop invariant inference can be found in Appx. B. Secs. 2.2, 2.3, and 2.4 extend on the techniques in Appx. B to unify class invariant inference, array verification, and parameterized compositional model checking under a single synthesis framework. Sec. 2.5 discusses the benefits of predicate templates and explains why IPS-MP requires both assumptions and assertions of partial predicates. We note that the automation in SmartACE is a special case of Sec. 2.3.

2.1 Methodology

In Figure 3, a single predicate (i.e., `Post`) represents a single unknown (i.e., the post-condition of a loop). This permits an IPS-MP solver to explore all relations between arguments (e.g. `x` and `y` of `Post`). When there are many variables, or large variable domains, the space of candidate solutions becomes very large. Restricting the syntactic structure of each unknown, referred to as its *shape*, helps to prune the search space. In general, an unknown can be split into cases (see Sec. 2.3), and the variables in each case can be partitioned (see Appx. B). Each partition is encoded by a unique predicate. Refining a predicate's shape prunes the candidate solution space, but may eliminate valid solutions.

Whenever an unknown is refined, the syntactic changes are reflected only where the unknown is assumed or asserted. The program remains unchanged otherwise. For this reason, in IPS-MP, it is convenient to separate unknowns from their shapes. In the context of program verification, this is accomplished with the following methodology. First, a proof-rule for the program of interest is reduced to assumptions and assertions on one or more unknowns. This is done once per proof-rule. Second, the shape of each unknown is refined using insight from the program. Third, the program is instrumented with assumptions and assertions. The instrumented program is an IPS-MP problem and is automatically solved by an IPS-MP solver. We illustrate this methodology using examples from object-oriented program analysis, array verification, and parameterized verification.

2.2 Class Invariant Inference as Synthesis

As a first example, we illustrate a reduction from class invariant inference to IPS-MP. In object-oriented programming, a class bundles together a data structure, its initialization procedure, and its operations. For example, the `Counter` class in Figure 4a accumulates values between 0 and some maximum value. The underlying data structure is a pair consisting of the current value, `pos`, and the maximum value, `max`. The initialization procedure on lines 3–6 first ensures that `_max` is positive, and then sets the current value to 0 and the maximum value to `_max`. The operations for `Counter` include `reset`, `capacity`, and `increment`. When `reset` is called, the current value is set back to 0. When `capacity` is called, the distance to the maximum value is returned. When `increment` is called, if `capacity` is greater than 0, then the current value is incremented and `true` is returned, else the current value is unchanged and `false` is returned.

The goal of this example is to prove that `drain` satisfies its assertions. The `drain` function takes an instance of `Counter` (in an arbitrary state), exhausts the counter's capacity, and then resets the counter to 0. The function is correct if `increment` always returns `true` on line 17, and `capacity` always returns a positive value on line 19. Verifying these claims

```
1  class Counter {
2    int max; int pos;
3    Counter(int _max) {
4      assume(_max > 0);
5      max = _max;
6      pos = 0; }
7    void reset() { pos = 0; }
8    int capacity() {
9      return max - pos; }
10   bool increment() {
11     if (pos >= max) return false;
12     pos += 1;
13     return true; }}
14
15 void drain(Counter o) {
16   while (o.capacity() > 0) {
17     assert(o.increment()); }
18   o.reset();
19   assert(o.capacity() > 0); }
```

(a) The original program.

```
1  bool PRED_TEMPLATE CInv(int m, int p) {
2    return synth(m, p); }
3  void main(void) {
4    if (*) {
5      Counter o(*);
6      assert(CInv(o.max, o.pos));
7    } else if (*) {
8      Counter o = *; assume(CInv(o.max, o.pos));
9      o.reset();
10     assert(CInv(o.max, o.pos));
11   } else if (*) {
12     Counter o = *; assume(CInv(o.max, o.pos));
13     o.increment();
14     assert(CInv(o.max, o.pos));
15   } else {
16     Counter o = *; assume(CInv(o.max, o.pos));
17     drain(o); }}
```

(b) The IPS-MP problem (using Figure 4a).

Figure 4 A program (see Figure 4a) which is correct relative to the choice of class invariant $(0 < \mathsf{o.max}) \wedge (0 \leq \mathsf{o.pos} \leq \mathsf{o.max})$, and a corresponding IPS-MP instance.

is non-trivial, as the correctness of `drain` depends on the possible states of `Counter`. For example, proving the assertion on line 17 requires the invariant $(0 \leq \mathtt{max} - \mathtt{pos})$.

A common approach to the modular analysis of object-oriented programs is class invariant inference (e.g., [24, 37, 1, 46]). A class invariant is a predicate that is true of a class instance after initialization, closed under the application of impure class methods, and sufficient to prove the correctness of the class [37]. In the case of `Counter`, the impure methods are `reset` and `increment`. Therefore, a class invariant for `Counter` must satisfy four requirements.

Figure 4b illustrates a technique to encode multiple cases in a single IPS-MP program. Intuitively, this program uses non-determinism to execute one of four possible cases. A case is selected on line 4 by a sequence of `if`-`else` statements, each with a non-deterministic condition. Even though the execution of each case is mutually exclusive, the IPS-MP solution must work in all cases. The cases in Figure 4b correspond to the requirements of a class invariant for `Counter`. To ensure that the class invariant holds after initialization, the first case initializes an instance of `Counter` with non-deterministic arguments, and then asserts that the instance satisfies the class invariant (lines 4–6). To ensure that the class invariant is closed with respect to `reset`, the second case selects an arbitrary instance of `Counter` (through non-determinism), assumes that this instance satisfies the class invariant, executes `reset`, and then asserts that the instance continues to satisfy the class invariant (lines 7–10). Similarly, the third case ensures that the class invariant is closed with respect to `increment` (lines 11–14). Finally, to ensure that the class invariant entails the correctness of `drain`, the fourth case selects an arbitrary instance of `Counter`, assumes that this instance satisfies the class invariant, and then calls `drain` with the instance as an argument (lines 15–17). This gives a program with unknowns, as required by the verification methodology.

Next, the shape of the class invariant is considered. In this example, we lack program-specific knowledge to help split the invariant into sub-cases. Furthermore, it would be futile to partition the invariant's arguments, as the invariant must relate `max` to `pos` (e.g., line 17 of Figure 4a requires that $0 \leq \mathtt{max} - \mathtt{pos}$). Therefore, $\mathtt{CInv}(m, p)$ is used as the shape of the invariant. In Figure 4b, `CInv` corresponds to the partial predicate on line 1. One solution to Figure 4b is the expression `(m > 0) && (p <= m)` for the hole in `CInv`. To prove the correctness of `drain`, a synthesizer may also infer the invariant $(0 \leq \mathtt{o.pos} \wedge \mathtt{o.pos} \leq \mathtt{o.max})$ for the loop on line 16 of Figure 4a.

```
1  void main(int sz) {
2    assume(sz > 0);
3    int *data = new int[sz];
4    memset(data, 0, sz * sizeof(int));
5    int max = 0; int sid = *;
6    assume(0 <= sid && sid < sz);
7    while (true) {
8      int id = *;
9      assume(0 <= id && id < sz);
10     int v = data[id];
11     if (id != sid) {v += 1;}
12     if (v > max) { max = v; }
13     data[id] = v; }}
```

(a) The original program.

```
1  bool PRED_TEMPLATE Inv3(int m, int v) {
2    if (m == 0 && v == 0) { return true; }
3    else { return synth(m, v); } }
4  bool PRED_TEMPLATE Inv4(int m, int v){
5    if (m == 0 && v == 0) { return true; }
6    else { return synth(m, v); }}
7  void main(int sid) {
8    int max = 0;
9    while (true) {
10     int id = *;
11     int x = *;
12     assume(id != x);
13     // int v = data[id];
14     int v = *; int u = *;
15     if (id == sid) { assume(Inv3(max,v)); }
16     else { assume(Inv4(max,v)); }
17     if (x == sid){ assume(Inv3(max,u)); }
18     else { assume(Inv4(max,u)); }
19     // Properties.
20     assert(v <= max);
21     if (id == sid) { assert(v == 0); }
22     // Update.
23     if (id != sid) { v += 1; }
24     if (v > max) { max = v; }
25     // data[id] = v;
26     if (id == sid) { assert(Inv3(max,v)); }
27     else { assert(Inv4(max,v)); }
28     if (x == sid) { assert(Inv3(max,u)); }
29     else { assert(Inv4(max,u)); }}}
```

(b) The IPS-MP problem.

Figure 5 A program (see Figure 5a) which is correct relative to the choice of array abstraction $(i = \mathsf{s} \wedge v = 0) \vee (i \neq \mathsf{s} \wedge 0 \leq v \leq \mathsf{max})$, and a corresponding IPS-MP instance.

2.3 Verification of Array-Manipulating Programs as Synthesis

Consider the array-manipulating program in Figure 5a. This program initializes the array `data`, and then performs an unbounded sequence of updates to the cells of `data` while maintaining the maximum element of `data` in `max`. A special index, stored by `sid`, remains unchanged during execution. On lines 2–4, `data` is allocated and then zero-initialized. On line 5, `max` is set to 0, since the maximum element of a zero-initialized array is 0. On line 6, `sid` is set to an arbitrary index in `data`. The unbounded sequence of updates begins on line 7, when the program enters a non-terminating loop. During each iteration, an index is selected (lines 8–9), and if this index is not `sid`, then the corresponding cell in `data` is incremented by 1 (line 11). If the cell is incremented, then `max` is updated accordingly (line 12). Note that Figure 5a can be thought of as a simplified smart contract, where `data` is an *address mapping*, `sid` is an *address variable*, and each iteration of the loop is a *transaction* . For a more general presentation of smart contracts as array-manipulating programs, see [67].

The goal of this example is to prove two properties about the cells of `data`. The first property is that every cell of `data` is at most `max`. The second property is that `data[sid]` is always zero. It is not hard to see why these properties are true. For example, the first property is true since every cell of `data` is initially zero, and after increasing the value of a cell, `max` is updated accordingly. However, verifying these properties is challenging, since `data` has an arbitrary number of cells. One solution to this problem is to compute a summary for each cell of `data`, with respect to `max` and `sid`, and independent of `data`'s length. This summary is then used in place of each array access to obtain a new program with a finite number of cells. For simplicity, we assume that array accesses are in bounds, and that integers cannot overflow (i.e., are modeled as mathematical integers).

A common approach to obtain such a summary is to over-approximate the least fixed

point of the program by an *abstract domain* that provides a tractable set of array cell partitions according to semantic properties (e.g., [29, 34, 20]). An alternative approach (followed here) is to compute a *compositional invariant* [50] for each cell of the array. A compositional (array) invariant is a predicate that is initially true of all cells in the array, and closed under every write to the array. Furthermore, a compositional invariant must be closed under *interference*, that is, if $i \neq j$ and the cell `data[i]` is updated, then `data[j]` continues to satisfy the compositional invariant. That is, a compositional invariant is assumed after each read and asserted after each write.

Using this approach, the program in Figure 5b is obtained. On line 10, an arbitrary index named `id` is selected, as in the original program. However, on lines 11–12, a second, *distinct* index named `x` is selected, to stand for a cell under interference. On lines 14–18, the compositional invariant is assumed, in place of reading the values at `data[id]` and `data[x]`. On lines 20–21, the two properties are asserted. If an arbitrary cell satisfies both properties, then every cell must satisfy both properties. On lines 23–24, the cell updates are performed as in the original program. On lines 26–29, the compositional invariant is asserted, in place of writing to `data[id]`. Note that lines 2–4 of Figure 5a do not appear in Figure 5b since the compositional invariant abstracts away the contents of `data`. This gives a program with unknowns, as required by the verification methodology.

Next, the shape of the compositional invariant is restricted. Observe that on line 11 of Figure 5a, the value written into `data` depends on whether the index is `sid`. This suggests that the compositional invariant has two cases that branch on whether `id` equals `sid`, namely $((\texttt{id} = \texttt{sid}) \wedge \texttt{Inv3}(\texttt{max}, v)) \vee ((\texttt{id} \neq \texttt{sid}) \wedge \texttt{Inv4}(\texttt{max}, v))$. In the IPS-MP encoding, both `Inv3` and `Inv4` correspond to partial predicates (see lines 1 and 4 in Figure 5b, respectively). The templates, on lines 2 and 5, correspond to the initialization rule for the invariant. Recall, however, that these templates are not strictly necessary. One alternative is to assert `Inv3(max,0)` and `Inv4(max,0)` before line 9, though this is not illustrated. Due to the branching shape of the invariant, each `assume` and `assert` statement must branch between the two partial predicates (see lines 14–18 and 26–29). Given Figure 5b, a synthesizer may find the expressions `(v == 0)` for the hole in `Inv3`, and `(0 <= v) && (v <= max)` for the hole in `Inv4`. By substitution, $((\texttt{id} = \texttt{sid}) \wedge (v = 0)) \vee ((\texttt{id} \neq \texttt{sid}) \wedge (0 \leq v) \wedge (v \leq \texttt{max}))$. To verify `main`, a synthesizer may also infer the invariant $(0 \leq \texttt{max})$ for the loop at line 9.

2.4 Parameterized Verification as Synthesis

As a third example, we illustrate a reduction from parameterized verification to IPS-MP. This example considers two or more processes running in a ring network of arbitrary size. A ring network organizes processes into a single cycle, such that each process has a left and right neighbour [19]. In this ring, adjacent processes share a lock on a common resource. Processes are either *trying* to acquire a lock, or have acquired all locks and are in a *critical* section. Initially, all processes are trying and all locks are free. Each processes runs the program in Figure 6a. The state of each process is given by `View` on line 3, and the transition relation of each process is given by `tr`[4] on line 5. Since each process runs the same program with the same configuration of locks, the ring network is said to be *symmetric*.

The goal of this example is to prove that if a process is in its critical section, then the process holds both adjacent locks. Following [50], this property is proven by computing an

[4] For simplicity, `tr` is not deadlock-free as processes retain their locks until reaching their critical sections. However, the critical section can be reached any number of times without encountering a deadlock.

```
typedef enum { Free, Left, Right } Lock;
typedef enum { Try, Critical } State;
struct View {
  Lock lhs; Lock rhs; State st; };
View tr(View v) {
  bool held = v.lhs == Left &&
              v.rhs == Right
  if (v.st == Critical) {
    v.st = Try;
    v.lhs = Free;
    v.rhs = Free; }
  else if (held) {
    v.st = Critical; }
  else if (v.lhs == Free) {
    v.lhs = Left; }
  else if (v.rhs == Free) {
    v.rhs = Right; }
  return v; }
```

(a) The process.

```
1  bool PRED_TEMPLATE RInv(
2    Lock l, State s, Lock r) {
3    if (l == Free && r == l && s == Try) {
4      return true;
5    } return synth(l, s, r); }
6  void main(struct View v) {
7    if (*) {
8      State otr = *;
9      assume(RInv(v.left, v.st, v.right));
10     assume(RInv(v.right, otr, v.left));
11     v = tr(v);
12     assert(RInv(v.left, v.st, v.right));
13     assert(RInv(v.right, otr, v.left));
14   } else {
15     assume(RInv(v.left, v.st, v.right));
16     bool held = v.left == Left &&
17                 v.right == Right;
18     assert(v.st != Critical || held); }}
```

(b) The IPS-MP problem (uses `tr`).

Figure 6 A process for a parameterized ring, and an IPS-MP problem that verifies the process. The process is correct relative to the compositional invariant $((\mathsf{v.lhs} \neq \mathsf{Left}) \vee (\mathsf{v.rhs} \neq \mathsf{Right})) \Rightarrow (\mathsf{v.st} \neq \mathsf{Critical})$, and the IPS-MP problem synthesizes the compositional invariant. Note that `Lock` and `State` are defined in Figure 6a using `typedef`, and that `otr` is a process under interference.

adequate compositional invariant for each process. An adequate compositional invariant is true of the initial state of each process, closed under the transition relation, closed under interference, and entails the property of interest. Remarkably, in a symmetric ring network, a compositional invariant can be computed by analyzing a ring with exactly two processes.

Using this approach, the program in Figure 6b is obtained. This program uses a non-deterministic `if` statement to split the analysis into two cases (line 7). The first case instantiates a two-process network using the compositional invariant (lines 8–10). Due to network symmetry, the left lock of the first process is the right lock of the second process, and vice versa. A single process in this network executes a transition (line 11), and then the closure of the compositional invariant is asserted for both processes (lines 12–13). The assertions on lines 12–13 ensure both closure under the transition relation and closure under interference, since only a single process transitioned. The second case instantiates a single process using the compositional invariant (line 15), and then asserts the property of interest (lines 16–18). Together, these cases define a compositional invariant. This gives a program with unknowns, as required by the verification methodology.

Next, the shape of the compositional invariant is considered. In this example, there is no motivation to split the invariant into cases. Furthermore, it would not make sense to partition the arguments of the invariant, since the state of a process is dependent on the combined state of its adjacent locks. Therefore, $\texttt{RInv}(l, s, r)$ is assumed to be the shape of the invariant. In the IPS-MP encoding, `RInv` corresponds to the partial predicate on line 1. The template on line 3 ensures that the compositional invariant is true of the initial state of each process. As an alternative to a template, one can instead assert `RInv(Free,Try,Free)` before line 7. One solution to this problem is to fill the hole in `RInv` with the expression `(s == Try) || ((l == Left) && (r == Right))`. Consequently, $((s \neq \texttt{Try}) \Rightarrow (l = \texttt{Left} \wedge r = \texttt{Right}))$.

2.5 Discussion

In Sec. 2.3, all explicit constraints were easily replaced by implicit constraints. However, explicit constraints can yield more succinct encodings. For example, consider the initial

```
bool PRED_TEMPLATE Inv(int x, int y) {
  if (x + y == 5) { return true; }
  else { return synth(x, y); } }
void main(void) { ... }
```

(a) Predicate template encoding.

```
bool PRED_TEMPLATE Inv(int x, int y) {
  return synth(x, y); }
void main(void) {
  int x = *; int y = *;
  assume(x + y == 5);
  assert(Inv(x, y));
  ... }
```

(b) Assertion-based encoding.

Figure 7 The initial condition ($x + y = 5$) encoded using a predicate (see Figure 7a), and its equivalent encoding using an assertion (see Figure 7b).

condition ($x + y = 5$). In Figure 7a, the condition is given as an explicit predicate template, and in Figure 7b, it is desugared as an assertion. To desugar the constraint, additional variables and assumptions are required.

In the examples presented so far, each IPS-MP problem places both assumptions and assertions on each partial predicate. All interesting IPS-MP problems follow this pattern. However, IPS-MP is well defined even if a partial predicate has only assumptions placed on it, only assertions placed on it, or neither. In these cases, the IPS-MP problem is trivial or reduces to a simpler problem.

If partial predicates only appear in assumptions, then the synthesized solution is never strengthened. In other words, the solution may be arbitrarily weak. This is an instance of specification synthesis. Usually, in specification synthesis, non-functional requirements are placed on each specification to ensure that a solution is "interesting" (e.g., [21, 2, 54]). Without these requirements, uninteresting solutions, such as `false`, are permitted. Since IPS-MP only places functional requirements on its solutions, this case is trivial and returning `false` from each predicate is always a solution (given a correct program).

If partial predicates only appear in assertions, then the synthesized solution is only ever strengthened. A solution in this case is an expression that subsumes all assertions placed on the predicate. However, an expression that evaluates to `true` subsumes all possible assertions. Therefore, this case is also trivial and returning `true` from each predicate is always a solution (given a correct program).

If partial predicates never appear in the program, then the synthesizer can select an arbitrary implementation for each predicate. However, if the synthesizer returns a solution, then the program must be correct relative to the solution. Therefore, if the program does violate an assertion, then the synthesizer must return a witness to unrealizability instead. Consequently, the output of the synthesizer indicates if the program is correct, and is equivalent to verification.

3 Background

This section recalls results from logic-based program verification. Sec. 3.1 reviews the key definitions of First Order Logic (FOL) and the Constrained Horn Clause (CHC) fragment of FOL. Sec. 3.2 introduces a programming language used throughout this paper. Sec. 3.3 recalls the connection between CHCs and program semantics through the weakest liberal precondition transformer.

3.1 First Order Logic and Constrained Horn Clauses

A *first order signature* Σ defines a set of predicates, a set of relations, and their respective arities. Given a set of variables $\mathcal{V}$, a *term* is either a variable from $\mathcal{V}$ or an application of

a relation in Σ to one or more terms. An *atom* is an application of a predicate in Σ to one or more terms. A *formula* joins atoms using standard logical connectives, existential quantification, and universal quantification. A formula is *quantifier-free* if it contains neither existential nor universal quantification. A formula is a *sentence* if all variable instances are quantified. Given a FO-formula φ, the formula $\varphi[x/y]$ is defined by substituting y for all free instances of x in φ. We write $\mathrm{Term}(\Sigma, \mathcal{V})$ and $\mathrm{QFFml}(\Sigma, \mathcal{V})$ for the sets of terms and quantifier-free formulas generated by Σ and $\mathcal{V}$.

A FO-*theory* $\mathcal{T}$ is a deductively closed set of sentences over a signature Σ. A $\mathcal{T}$*-model* for a formula φ is an interpretation of each predicate, relation, and free variable in $\mathcal{T} \cup \{\varphi\}$ such that every formula in $\mathcal{T} \cup \{\varphi\}$ is true. If a $\mathcal{T}$-model exists for φ, then φ is *satisfiable*, otherwise, φ is *unsatisfiable*. In the case that all valid interpretations of $\mathcal{T}$ are $\mathcal{T}$-models for φ, then φ is $\mathcal{T}$-valid and we write $\models_{\mathcal{T}} \varphi$. Furthermore, if each interpretation of a $\mathcal{T}$-model M can be expressed in some logical fragment $\mathcal{F}$, then M provides an $\mathcal{F}$*-solution* to φ .

Constrained Horn Clauses (CHCs) are a fragment of FOL determined by a FO-signature Σ and an set of predicates P. A CHC is a sentence of the form $\forall V \cdot \varphi \wedge p_1(\vec{x}_1) \wedge \cdots \wedge p_k(\vec{x}_k) \Rightarrow h(\vec{y})$, where $\varphi \in \mathrm{QFFml}(\Sigma, \mathcal{V})$ and $\{p_1, \ldots, p_k, h\} \subseteq P$. For program semantics, it is useful to use v' to denote the value of a variable v after a program transition and $\mathsf{keep}(W) := \bigwedge_{w \in W} w = w'$ to denote that variables $W \subseteq \mathcal{V}$ are unchanged during a transition. Given a set of variables $V = \{v_1, \ldots, v_n\} \subseteq \mathcal{V}$, the set of variables $\{v'_1, \ldots, v'_n\}$ is denoted V'. Likewise, given a formula φ over the variables in V, the formula $\varphi[v_1/v'_1] \cdots [v_n/v'_n]$ over V' is denoted φ'.

3.2 Procedural Programming Language

Throughout this paper, we consider a simple procedural programming language, whose syntax is standard and can be found in Appx. A. We assume that all expressions are factored out by a FO-signature Σ, with variables from a set $\mathcal{V}$. That is, each expression is of the form $\mathrm{QFFml}(\Sigma, \mathcal{V})$. The set of all programs in the language is denoted $\mathsf{Progs}(\Sigma, \mathcal{V})$. For simplicity, types are omitted. In this language, a program has one or more procedures, with execution starting from `main`. Each procedure is written in an imperative language, including loops and procedure calls. The language is extended with a non-deterministic assignment (i.e., `*`), and verification statements `assume` and `assert`. The expressions in assume and assert can be either from $\mathrm{QFFml}(\Sigma, \mathcal{V})$ or a call to a pure Boolean procedure, called a *predicate*. Predicates may only be called within assume or assert statements. Given a program $\mathcal{P} \in \mathsf{Progs}(\Sigma, \mathcal{V})$, $\mathsf{Procs}(\mathcal{P})$ denotes the procedures in $\mathcal{P}$. A special case is when all variables are Boolean.

▶ **Definition 1.** *Let Σ_{Bool} denote a Boolean signature. A* Boolean program *is a tuple $(Locs, GV, LV, E)$ with $E = (NE, CE, FE, AE, PE)$ and $V = GV \cup LV$ such that:*

1. *$Locs$ is a finite set of control-flow locations with entry-point* `main` $\in Locs$*;*
2. *GV and LV are disjoint sets of local and global variables (respectively);*
3. *$NE \subseteq Locs \times \mathrm{QFFml}(\Sigma_{\mathrm{Bool}}, V \cup V') \times Locs$ is a set of* normal edges*, $CE \subseteq Locs \times Locs \times Locs$ is a set of* call edges*, $FE \subseteq Locs \times Locs \times Locs$ is a set of* (partial predicate) call-under-assume edges*, $AE \subseteq Locs \times Locs \times Locs$ is a set of* (partial predicate) call-under-assert edges*, and $PE \subseteq Locs \times Locs$ is a set of* procedure summary edges*;*
4. *If $(l_1, R, l_2) \in NE$, then l_2 is reachable from l_1 by updating the variables according to R and if $(l_{\mathrm{call}}, l_{\mathrm{in}}, l_{\mathrm{ret}}) \in CE$, then l_{ret} is reachable from l_{call} by executing the procedure with entry location l_{in};*
5. *If $(l_{\mathrm{call}}, l_{\mathrm{in}}, l_{\mathrm{ret}}) \in FE$, then l_{ret} is reachable from l_{in} by assuming the partial predicate with entry location l_{call} and if $(l_{\mathrm{call}}, l_{\mathrm{in}}, l_{\mathrm{ret}}) \in AE$, then l_{ret} is reachable from l_{in} by asserting the partial predicate with entry location l_{call};*

$$\mathsf{ToCHC}(\mathcal{P}) := wlp(\mathcal{P}(\mathtt{Main}), \top) \wedge \left(\bigwedge_{f \in \mathsf{Procs}(\mathcal{P})} \mathsf{ToCHC}(f) \right)$$

$$\mathsf{ToCHC}\left(f(\vec{x})\ \{\ S;\ \mathsf{return}\ \vec{e};\ \}\right) := \forall \overline{\vec{x}} \cdot \left(\vec{x} = \overline{\vec{x}} \wedge f_{pre}(\vec{x}) \Rightarrow wlp\left(S, f_{sum}\left(\overline{\vec{x}}, \vec{e}\right)\right)\right)$$

$$\mathsf{ToCHC}\left(p(\vec{x})\ \{\ \mathsf{return}\ e;\ \}\right) := \forall \vec{x} \cdot p(\vec{x}) \Leftrightarrow e$$

Figure 8 The partial correctness conditions for a program $\mathcal{P} \in \mathsf{Progs}(\Sigma, \mathcal{V})$. This follows the presentation of [14].

6. *If $(l_{\text{in}}, l_{\text{out}}) \in PE$, then the procedure with entry location l_{in} has exit location l_{out}.*

A Boolean program consists of control-flow locations and edges between locations. Each procedure has a single entry location, l_{in}, and a single exit location, l_{out}, where $(l_{\text{in}}, l_{\text{out}}) \in PE$. The program enters at `main` $\in$ *Locs*, and a special location $l_\perp \in$ *Locs* indicates failure. *Normal edges* connect control-flow locations within a procedure and represent non-procedural statements. For example, the statement `assert(e)` (where e is an expression) corresponds to two normal edges, $(l_1, e \wedge \mathsf{keep}(GV \cup LV), l_2)$ and $(l_1, \neg e \wedge \mathsf{keep}(GV \cup LV), l_\perp)$. *Call edges* (optionally under assume or assert) connect locations in a caller's procedure and a callee's procedure by giving the call and return locations of the caller (l_{call} and l_{ret}, respectively), and the entry location for the callee (l_{in}). For simplicity, all procedures have the same local variables, and arguments are passed by global variables. The location $l_\perp$ is assumed to have no outgoing edges.

The state of a Boolean program is a tuple (l, s), where l is a location and s is an assignment to each Boolean variable. Initially, $l =$ `main` and s is an arbitrary assignment. For each normal edge (l_1, R, l_2), a transition exists from (l_1, s_1) to (l_2, s_2), if $s_1 \wedge R \wedge s_2'$ is valid. All call edges have the expected semantics.

3.3 Logical Program Verification

The Weakest Liberal Precondition (WLP) transformer gives logical semantics to imperative programs [23]. We write $wlp(S, Q)$ for the WLP of a statement S with respect to a post-condition$Q$. The WLP transformer for $\mathsf{Progs}(\Sigma, \mathcal{V})$ is standard and can be found in Appx. A. Note that in this transformation, the $loop_{ln}$ predicate is an invariant for a loop at line ln.

The $wlp(-)$ transformer can be used to verify partial correctness for procedural programs. This is achieved through the $\mathsf{ToCHC}(-)$ transformer in Figure 8. The $wlp(\mathcal{P}(\mathtt{main}), \top)$ term asserts that `main` satisfies all assertions. For each procedure $f \in \mathsf{Procs}(\mathcal{P})$, the term $\mathsf{ToCHC}(f)$ asserts that f is correct for all inputs passed to f in every execution. Note that in Figure 8, f_{pre} collects inputs to f, and f_{sum} relates the inputs of f to the outputs of f. In the case that f is a predicate, f_{pre} and f_{sum} are omitted, since f is side-effect free. Together, $\mathsf{ToCHC}(\mathcal{P})$ asserts that the program $\mathcal{P}$ is correct for any execution starting from `main`. If $\mathsf{ToCHC}(\mathcal{P})$ is satisfiable, then there exist loop invariants for $\mathcal{P}$ such that $\mathcal{P}$ satisfies all assertions [14]. Therefore, $\mathsf{ToCHC}(\mathcal{P})$ can be used to verify $\mathcal{P}$. Furthermore, $\mathsf{ToCHC}(\mathcal{P})$ is in the CHC fragment [14].

Efficient procedures exist to prove that Boolean programs are correct. For example, *program summarization* simultaneously computes a summary θ from control-flow locations to input-to-reachable-state relations, and a summary σ from procedures to input-output relations. For a location $l \in$ *Locs*, if $\theta(l) = \perp$, then l is unreachable. Therefore, a Boolean program $\mathcal{P}$ is correct if and only if $\theta(l_\perp) = \perp$ in the least summary of $\mathcal{P}$ [6]. Program

Algorithm 1 Computes the least Boolean program summary (θ, σ) [8]. Follows the presentation of [17].

```
1  var (θ, σ) ;                          // A program summary
2  var W ; // A map from Locs to a queued state
3  Func UpdateReach(l, v):
4      s_diff ← v ∧ ¬θ(l);
5      if s_diff ≠ ⊥ then
6          θ(l) ← θ(l) ∨ s_diff;
7          W(l) ← W(l) ∨ s_diff
8  Func DoIntraproc(V, NE, l_wk, s_wk):
9      for (l_wk, R, l_2) ∈ NE do
10         s_2 ← elim(s_wk ∧ R*, V');
11         s_2 ← s_2[V''/V'];
12         UpdateReach(l_2, s_2);
13 Func DoProcSum(V, LV, PE, CE, l_wk, s_wk):
14     for (l_in, l_wk) ∈ PE do
15         s_sum ← elim(s_wk, LV ∪ LV') ∧ ¬σ(l_in);
16         if s_sum = ⊥ then continue;
17         σ(l_in) ← σ(l_in) ∨ s_sum;
18         for (l_call, l_in, l_ret) ∈ CE do
19             X ← s*_sum ∧ keep(LV') ;
20             s ← elim(θ(l_call) ∧ X, V')[V''/V'];
21             UpdateReach(l_ret, s);
22 Func DoProcs(V, LV, PE, CE, l_wk, s_wk):
23     for (l_wk, l_in, l_ret) ∈ CE do
24         s_in ← elim(s_wk, V ∪ LV')[V'/V];
25         UpdateReach(l_in, s_in);
26         X ← σ(l_in)* ∧ keep(LV');
27         s ← elim(s_wk ∧ X, V')[V''/V'];
28         UpdateReach(l_ret, s);
29 Func InitBoolReach(Locs, PE):
30     for l ∈ Locs do θ(l) ← ⊥ ;
31     for (l_in, l_out) ∈ PE do σ(l_in) ← ⊥ ;
32     θ(main) ← ⊤; W(main) ← ⊤;
33 Func ComputeBoolReach(𝒫):
34     (Locs, GV, LV, (N, C, ∅, ∅, P)) ← 𝒫;
35     V ← GV ∪ LV;
36     InitBoolReach(Locs, P);
37     while ∃ l_wk ∈ Locs · W(l_wk) ≠ ⊥ do
38         s_wk ← W(l_wk) ; W(l_wk) ← ⊥;
39         DoIntraproc(V, N, l_wk, s_wk);
40         DoProcs(V, LV, P, C, l_wk, s_wk);
41         DoProcSum(V, LV, P, C, l_wk, s_wk);
```

summarization is defined in Def. 2[5]. The algorithm to compute θ is presented in full, for reuse in Sec. 5.1. For presentation, $\mathsf{elim}(\varphi, W)$ denotes the existential elimination of W in φ.

▶ **Definition 2** ([6]). *A* Boolean program summary *for* $(Locs, GV, LV, E)$*, where* $E = (NE, CE, \varnothing, \varnothing, PE)$ *is a tuple* (θ, σ) *such that* $Q = \mathrm{QFFml}(\Sigma_{\mathrm{Bool}}, V \cup V')$*,* $V = GV \cup LV$ *and the following hold:*

1. $\sigma : Locs \to Q$ *and* $\theta : Locs \to Q$*;*
2. $\sigma(\texttt{main}) = \top$*;*
3. $\forall (l_1, R, l_2) \in NE \cdot \theta(l_1) \wedge R' \Rightarrow \theta(l_2)[V'/V'']$*;*
4. $\forall (l_{\mathrm{call}}, l_{\mathrm{in}}, l_{\mathrm{ret}}) \in CE \cdot \theta(l_{\mathrm{call}}) \wedge \sigma'(l_{\mathrm{in}}) \wedge \mathit{keep}(LV') \Rightarrow \theta(l_{\mathrm{ret}})[V'/V'']$*;*
5. $\forall (l_{\mathrm{call}}, l_{\mathrm{in}}, l_{\mathrm{ret}}) \in CE \cdot \mathit{elim}(\theta(l_{\mathrm{call}}), V \cup LV') \Rightarrow \theta(l_{\mathrm{in}})$*;*
6. $\forall (l_{\mathrm{in}}, l_{\mathrm{out}}) \in PE \cdot \theta(l_{\mathrm{out}}) \Rightarrow \sigma(l_{\mathrm{in}})$*.*

`ComputeBoolReach` in Algorithm 1 is the standard algorithm to compute a least program summary. The algorithm works by iteratively applying the rules of Def. 2 until a fixed point is reached (we write $R^* := R[V'/V''][V/V']$). Termination is ensured by the finite-state of Boolean programs and the monotonicity of each rule. We extend on the algorithm `ComputeBoolReach` in Sec. 5.1.

4 IPS-MP: Problem Definition

This section defines partial predicates and the IPS-MP problem. A *partial predicate* is a pure Boolean function without an implementation. A program $\mathcal{P}$ is *open* if it contains a partial predicate p. An implementation for p is a Boolean expression e over the arguments of p. The program obtained by implementing p as `return` e is denoted $\mathcal{P}[p \leftarrow e]$. The set of all partial

[5] To align with Algorithm 1, Def. 2 is non-standard but equivalent to [6].

```
1 bool Post(int x, int y) {
2   return x==y; }
3 void main(int y) {
4   assume(y>0); int x=0;
5   for (int i=0; i<y; ++i) { x+=1; }
6   assert(Post(x, y));
7   x=*; y=*; assume(Post(x, y)); }
```

(a) The program $\mathcal{P}[\Pi_{weak}]$.

```
1 bool Post(int x, int y) {
2   return (y>0) && (x==y); }
3 void main(int y) {
4   assume(y > 0); int x=0;
5   for (int i=0; i<y; ++i) { x+=1; }
6   assert(Post(x, y));
7   x=*; y=*; assume(Post(x, y)); }
```

(b) The program $\mathcal{P}[\Pi_{strong}]$.

Figure 9 Implementations of the simple program in Figure 3.

predicates in $\mathcal{P}$ is written $\mathsf{Partial}(\mathcal{P}) = \{p_1, \ldots, p_k\}$. Given a function Π from $\mathsf{Partial}(\mathcal{P})$ to pure Boolean expressions, we write $\mathcal{P}[\Pi]$ to denote $\mathcal{P}[p_1 \leftarrow \Pi(p_1)] \cdots [p_k \leftarrow \Pi(p_k)]$. The IPS-MP problem is to find a Π such that $\mathcal{P}[\Pi]$ is correct.

▶ **Example 3.** Recall program $\mathcal{P}$ from Figure 3. Since `Post` is unimplemented in $\mathcal{P}$, then $\mathcal{P}$ is an open program. Formally, $\mathsf{Partial}(\mathcal{P}) = \{\texttt{Post}\}$. In Sec. 2, two implementations were proposed for `Post`, namely $(x = y)$ and $(y > 0 \wedge x = y)$. These implementations are represented by the mappings Π_{weak} and Π_{strong} from $\mathsf{Partial}(\mathcal{P})$ to pure Boolean expressions such that $\Pi_{weak} : \texttt{Post} \mapsto (x = y)$ and $\Pi_{strong} : \texttt{Post} \mapsto (y > 0 \wedge x = y)$. The closed programs $\mathcal{P}[\Pi_{weak}]$ and $\mathcal{P}[\Pi_{strong}]$ are illustrated in Figure 9a and Figure 9b, respectively. ◀

▶ **Definition 4.** *An* Inductive Predicate Synthesis Modulo Programs (IPS-MP) *problem is a tuple* $(\mathcal{P}, \mathcal{T}, \Pi_0)$ *such that* $\mathcal{P} \in \mathsf{Progs}(\Sigma, \mathcal{V})$ *with first-order signature* Σ *and variable set* $\mathcal{V}$, $\mathcal{T}$ *is a first-order theory, and* $\Pi_0 : \mathsf{Partial}(\mathcal{P}) \to \mathrm{QFFml}(\Sigma, \mathcal{V})$ *are predicate templates. A* solution *to* $(\mathcal{P}, \mathcal{T}, \Pi_0)$ *is a function* $\Pi : \mathsf{Partial}(\mathcal{P}) \to \mathrm{QFFml}(\Sigma, \mathcal{V})$ *such that* $\mathcal{P}[\Pi]$ *is correct relative to* $\mathcal{T}$ *and* $\forall p \in \mathsf{Partial}(\mathcal{P}) \cdot \models_{\mathcal{T}} \Pi_0(p) \Rightarrow \Pi(p)$.

Assume that $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is an IPS-MP problem with a solution Π. With respect to the IPS-MP overview in Figure 2, $\mathcal{P}$ is a *program with specifications*, Π_0 is a collection of *predicate templates*, and Π is an *implementation of partial predicates*. The *witness to unrealizability* is discussed in Sec. 5. As an example of Def. 4, Figure 5b is restated as a formal IPS-MP problem.

▶ **Example 5.** This example restates Figure 5b as an IPS-MP problem $(\mathcal{P}, \Pi_0, \mathcal{T})$. The program $\mathcal{P}$ is given by lines 7–29 of Figure 5b. Then $\mathsf{Partial}(\mathcal{P}) = \{\texttt{Inv3}, \texttt{Inv4}\}$, since `Inv3` and `Inv4` are called on lines 15–16, but lack full implementations. From lines 1–6, $\Pi_0(\texttt{Inv3}) = \Pi_0(\texttt{Inv4}) = (m = 0 \wedge v = 0)$. Now, recall from Sec. 2.3 that all variables in Figure 5b are arithmetic integers. Therefore, $\mathcal{T}$ is the theory of integer linear arithmetic. A solution to $(\mathcal{P}, \Pi_0, \mathcal{T})$ is Π such that $\Pi(\texttt{Inv3}) = (v = 0)$ and $\Pi(\texttt{Inv4}) = (0 \leq v \wedge v \leq m)$. ◀

5 Decidability of IPS-MP

This section considers the decidability of IPS-MP. Sec. 5.1 shows that IPS-MP is efficiently decidable in the Boolean case. Sec. 5.2 shows that IPS-MP is undecidable in general, but admits sound proof-rules for realizability and unrealizabiliy.

5.1 The Case of Boolean Programs

This section shows that for Boolean programs, IPS-MP is decidable with the same time complexity as problem verification (i.e., polynomial in the number of program states). In contrast, general synthesis is known to have exponential time complexity in the Boolean

case [65]. Therefore, IPS-MP modulo Boolean programs does in fact offer the benefits of general synthesis without the associated costs. To prove this result, we first extend Boolean program summaries (Def. 2) to programs with partial predicates. These new summaries are then used to extract solutions to IPS-MP (or witnesses to unrealizability). `Analyze` of Algorithm 2 extends on Algorithm 1 to compute these new summaries. The total correctness and time complexity of `Analyze` are proven in Cor. 9 and Thm. 7, respectively.

To simplify our presentation, we assume that all predicates are partial. In a Boolean program, each partial predicate has an entry location, but no edges nor exit location. This means that a standard summary can be obtained for a Boolean program with partial predicates by discarding all calls to partial predicates. Such a summary characterizes reachability, under the assumption that partial predicates are never called. From this summary, the arguments passed to each partial predicate under assert can be collected. For the program summary to be correct, the partial predicates must return true on these asserted arguments. If the partial predicate returns true on these asserted arguments, then for any call under assume using the same arguments, the program execution must continue to the next state. This procedure can then be repeated until a fixed point is obtained. This new *partial program summary* is defined formally in Def. 6.

▶ **Definition 6.** *Let $\mathcal{P} = (Locs, GV, LV, (NE, CE, FE, AE, PE))$ be a Boolean program. A* partial program summary *for $\mathcal{P}$ is a tuple (θ, σ, Π) such that:*

1. $\Pi : \mathit{Partial}(\mathcal{P}) \to \mathrm{QFFml}(\Sigma_{\mathrm{Bool}}, GV)$*;*
2. (θ, σ) *is a program summary for* $(Locs, GV, LV, (NE, CE, \varnothing, \varnothing, PE))$*;*
3. $\forall (l_{\mathrm{call}}, l_{\mathrm{in}}, l_{\mathrm{ret}}) \in AE \cdot \theta(l_{\mathrm{call}}) \Rightarrow \Pi'(l_{\mathrm{in}})$*;*
4. $\forall (l_{\mathrm{call}}, l_{\mathrm{in}}, l_{\mathrm{ret}}) \in AE \cdot \theta(l_{\mathrm{call}}) \Rightarrow \theta(l_{\mathrm{ret}})$*;*
5. $\forall (l_{\mathrm{call}}, l_{\mathrm{in}}, l_{\mathrm{ret}}) \in FE \cdot \theta(l_{\mathrm{call}}) \wedge \Pi'(l_{\mathrm{in}}) \Rightarrow \theta(l_{\mathrm{ret}})$.

The rules of Def. 6 follow directly from the preceding discussion. Rule 2 ensures that (θ, σ) is a program summary for $\mathcal{P}$ after discarding all calls to partial predicates. Rules3 and 4 collect the arguments passed to partial predicates under assert. Rule 5 advances the program state from calls to partial predicates under assume, according to the collected arguments. These steps are made operational by `Analyze` of Algorithm 2. Note that `Analyze` does not call `ComputeBoolReach` directly, and instead applies all rules in a single loop.

The termination of `Analyze` follows analogously to `ComputeBoolReach`. First, note that `Analyze` terminates if all work items have been processed. Each iteration of the loop at line 22 processes at least one work item. A state is added to the work list only if it has not yet been visited. The number of states is finite, since Boolean programs are finite-state. Therefore, `Analyze` must terminate with time polynomial in the number of program states. This is in contrast to general synthesis, which requires time exponential in the number of program states [65].

▶ **Theorem 7.** *Let $\mathcal{P} = (Locs, GV, LV, E)$ with $E = (NE, CE, FE, AE, PE)$ be a Boolean program. Then for each input $(\mathcal{P}, \Pi_0)$,* `Analyze` *of Algorithm 2 terminates in $O(n^2 m \cdot |Locs|)$ symbolic Boolean operations where $n = 2^{|GV \cup LV|}$ is the number of variable assignments and $m = \cdot |NE \cup CE \cup FE \cup AE|$ is the number of edges.*

The correctness of `BoolSynth` follows from the correctness of `ComputeBoolReach` in [17]. Thm. 8 proves that `Analyze` extends `ComputeBoolReach` to obtain a least partial program summary. Cor. 9 proves that an IPS-MP solution (or a witness to unrealizability) can be extracted from a least partial program summary. Since `Analyze` terminates, this is a decision procedure for the Boolean case of IPS-MP.

Algorithm 2 An extension of Algorithm 1 to solve IPS-MP for Boolean programs.

```
var (θ, σ, Π) ;      // A partial program summary
var W ;   // A map from Locs to queued states
Func DoAssumes(V, LV, PE, FE, l_wk, s_wk):
    for (l_wk, l_in, l_ret) ∈ FE do
        s_in ← elim(s_wk, V ∪ LV')[V'/V];
        UpdateReach(l_ret, Π(l_in) ∧ s_in);
Func DoAsserts(V, LV, PE, AE, l_wk, s_wk):
    for (l_wk, l_in, l_ret) ∈ AE do
        UpdateReach(l_ret, Π(l_in) ∧ s_wk) ;
        UpdateReach(l_in,
          elim(s_wk, V ∪ LV')[V'/V]);
Func DoFuncSum(V, LV, FE, AE, l_wk, s_wk):
    if l_wk ∈ Partial(P) then
        Π(l_wk) ← Π(l_wk) ∨ s_wk;
        for (l_call, l_wk, l_ret) ∈ FE ∪ AE do
            UpdateReach(l_ret, θ(l_call) ∧ s_wk)
Func Init(Locs, PE, Π_0):
    InitBoolReach(Locs, PE);
    for l ∈ Partial(P) do  Π(l) ← Π_0(l) ;
Func Analyze(P, Π_0):
    (Locs, GV, LV, (N, C, F, A, P)) ← P;
    V ← GV ∪ LV ; Init(Locs, P, Π_0);
    while ∃ l_wk ∈ Locs · W(l_wk) ≠ ⊥ do
        s_wk ← W(l_wk) ; W(l_wk) ← ⊥;
        DoIntraproc(V, N, l_wk, s_wk);
        DoProcs(V, LV, P, C, l_wk, s_wk);
        DoAssumes(V, LV, P, F, l_wk, s_wk);
        DoAsserts(V, LV, P, A, l_wk, s_wk);
        DoProcSum(V, LV, P, C, l_wk, s_wk);
        DoFuncSum(V, LV, F, A, l_wk, s_wk);
Func BoolSynth(P, Π_0):
    Analyze(P, Π_0);
    if θ(l_⊥) = ⊥ then  return (✓, Π) ;
    else  return (×, Π) ;
```

▶ **Theorem 8.** *Let $\mathcal{P} = (Locs, GV, LV, E)$ be a Boolean program and Π_0 be a collection of predicate templates for $\mathcal{P}$.* `Analyze` *of Algorithm 2 computes a least partial program summary, (θ, σ, Π), for $\mathcal{P}$ such that $\forall p \in \mathsf{Partial}(\mathcal{P}) \cdot \Pi_0(p) \Rightarrow \Pi(p)$.*

▶ **Corollary 9.** `BoolSynth` *of Algorithm 2 decides IPS-MP for Boolean programs.*

5.2 The General Case

This section presents sound proof-rules for the realizability and unrealizability of IPS-MP problems. These rules are shown to be instances of CHC-solving. To justify the reduction from IPS-MP to this undecidable problem, the general case of IPS-MP is also shown to be undecidable. First, assume that $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is an IPS-MP problem. Recall that $\mathcal{P} \in \mathsf{Progs}(\mathcal{F}, \mathcal{V})$ where $\mathcal{F}$ is the FO-fragment of pure program expressions. A logical encoding of $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is given by:

$$\mathsf{CHCSynth}(\mathcal{P}, \Pi_0) := \mathsf{ToCHC}(\mathcal{P}) \wedge \left(\bigwedge_{p \in \mathsf{Partial}(\mathcal{P})} \forall \vec{x} \cdot (\Pi_0(p) \Rightarrow p(\vec{x})) \right)$$

The term $\mathsf{ToCHC}(\mathcal{P})$ encodes verification conditions for $\mathcal{P}$, in which each partial predicate is unspecified. Calls to a partial predicate p, under assume and assert, provide constraints on the strongest and weakest possible solutions to $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$. The clause $\forall \vec{x} \cdot (\Pi_0(p) \Rightarrow p(\vec{x}))$ then ensures the strongest solution to p subsumes $\Pi_0(p)$. Then a solution σ to $\mathsf{CHCSynth}(\mathcal{P}, \Pi)$ contains an implementation $\sigma(p)$ for each partial predicate p, that subsumes $\Pi_0(p)$ and ensures the correctness of $\mathcal{P}$ (Thm. 10). Furthermore, if σ is an $\mathcal{F}$-solution, then each $\sigma(p)$ can be implemented in the programming language. On the other hand, if $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ is unsatisfiable, then for every choice of implementation Π satisfying Π_0, the closed program $\mathcal{P}[\Pi]$ is incorrect (Thm. 11). Together, these theorems give sound proof rules for the realizability and unrealizability of $(\mathcal{P}, \mathcal{T}, \Pi_0)$. In practice, $\mathcal{F}$ is chosen to be the same fragment used by the CHC-solver.

▶ **Theorem 10.** *Let Σ be a first-order signature, $\mathcal{V}$ be a set of variable symbols, $\mathcal{F} = \mathrm{QFFml}(\Sigma, \mathcal{V})$, $\mathcal{P} \in \mathsf{Progs}(\Sigma, \mathcal{V})$, and $(\mathcal{P}, \mathcal{T}, \Pi_0)$ be an IPS-MP problem. If σ is an $\mathcal{F}$-solution*

to $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ *relative to* $\mathcal{T}$*, then* $\Pi : \mathsf{Partial}(\mathcal{P}) \to \mathcal{F}$ *such that* $\Pi : p \mapsto \sigma(p)$ *is a solution to* $(\mathcal{P}, \mathcal{T}, \Pi_0)$*.*

▶ **Theorem 11.** *If* $(\mathcal{P}, \mathcal{T}, \Pi_0)$ *is an IPS-MP problem and* $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ *is* $\mathcal{T}$*-unsatisfiable, then* $(\mathcal{P}, \mathcal{T}, \Pi_0)$ *is unrealizable.*

$\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ strengthens $\mathsf{ToCHC}(\mathcal{P})$ by adding additional CHCs. Since $\mathsf{ToCHC}(\mathcal{P})$ is a conjunction of CHCs, then $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ is also a conjunction of CHCs. Therefore, a CHC solver can check the satisfiability and unsatisfiability of $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$. As a result, a CHC solver can find a solution to $(\mathcal{P}, \mathcal{T}, \Pi_0)$ (Thm. 10), or prove that the problem is unrealizable (Thm. 11).

▶ **Theorem 12.** $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ *is a CHC conjunction.*

▶ **Example 13.** This example uses Thm. 10 to solve the IPS-MP problem in Figure 4b. The program in Figure 4b corresponds to the IPS-MP problem $(\mathcal{P}, \mathcal{T}, \Pi_0)$ where $\mathcal{P}$ is the source code, $\mathcal{T}$ is the theory of integer linear arithmetic, and $\Pi_0 : \texttt{CInv} \to \bot$. In this example we let $\mathcal{F}$ be the fragment of linear inequalities of the variables $\{\mathsf{m}, \mathsf{p}\}$, where m and p are the arguments to `CInv`. Then our goal is to find an expression $e \in \mathcal{F}$ such that $\mathcal{P}[\texttt{CInv} \leftarrow e]$ is correct. According to Thm. 10, we can extract e from the output of a CHC-solver. The first step in this process is to construct the input $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$. To construct $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ we must first construct the term $\mathsf{ToCHC}(\mathcal{P})$. Recall that $\mathsf{ToCHC}(\mathcal{P})$ encodes verification conditions for the program $\mathcal{P}$. Since $\mathcal{P}$ is open (`CInv` is unimplemented), then `CInv` will be an unknown in $\mathsf{ToCHC}(\mathcal{P})$. According to Sec. 3.3, $\mathsf{ToCHC}(\mathcal{P})$ will consist of the verification conditions for $\mathcal{P}[\texttt{main}]$, along with a summary for each function in $\mathcal{P}$. We begin by constructing a summary for each method from the `Counter` object in $\mathcal{P}$. As described in Sec. 3.3, each predicate $f_{pre}(\mathbf{x})$ collects the inputs $\mathbf{x}$ to a function f, and each predicate $f_{sum}(\mathbf{x}, \mathbf{e})$ each argument $\mathbf{x}$ to a return value $\mathbf{e}$ For simplicity, we encode object state by passing member fields as arguments and return values. Redundant declarations are omitted.

$$
\begin{aligned}
\varphi_{Ctor} &:= \forall m \cdot Counter_{pre}(m) \Rightarrow ((m > 0) \Rightarrow Counter_{sum}(m, m, 0)) \\
\varphi_{Reset} &:= \forall m \cdot \forall p \cdot reset_{pre}(m, p) \Rightarrow reset_{post}(m, p, m, 0) \\
\varphi_{Cap} &:= \forall m \cdot \forall p \cdot capacity_{pre}(m, p) \Rightarrow capacity_{sum}(m, p.m - p \neq 0) \\
\varphi_{Incr} &:= \forall m \cdot \forall p \cdot increment_{pre}(m, p) \Rightarrow (\,((p \geq m) \Rightarrow increment_{sum}(m, p, \bot)) \wedge \\
&\qquad\qquad\qquad\qquad ((p < m) \Rightarrow increment_{sum}(m, p + 1, \top)))
\end{aligned}
$$

Next, we construct a summary for the function `drain`. Note that, unlike the methods of `Counter`, the function `drain` contains a loop. As described in Sec. 3.3, loops are encoded using loop invariants with the loop at line n associated with an invariant $loop_n$. In our example, the loop at Line 16 of Figure 4a is associated with a loop invariant $loop_{13}$. Then the summary of `drain` is as follows.

$$
\begin{aligned}
\varphi_{Exit} &:= capacity_{pre}(p', m') \cdot \forall x \cdot \Big(capacity_{sum}(p', m', x) \Rightarrow \Big(x \wedge drain_{sum}(p, m, p', m')\Big)\Big) \\
\varphi_{Loop} &:= loop_{13}(p, m, x) \wedge \\
&\quad ((loop_{13}(p, m, x) \wedge x > 0) \Rightarrow (capacity_{pre}(p, m) \wedge \forall x \cdot (capacity_{sum}(p, mx) \Rightarrow loop_{13}))) \wedge \\
&\quad ((loop_{13}(p, m, x) \wedge x \leq 0) \Rightarrow reset_{pre}(p, m) \wedge \forall p' \cdot \forall m' \cdot \Big(reset_{sum}(p, m, p', m') \Rightarrow \varphi_{Exit}\Big)) \\
\varphi_{Dr} &:= \forall p \cdot \forall m \cdot drain_{pre}(p, m) \Rightarrow (capacity_{pre}(p, m) \wedge \forall x \cdot (capacity_{sum}(p, m, x) \Rightarrow \varphi_{Loop}))
\end{aligned}
$$

Finally, we construct the verification conditions for `main`. Since `main` is the entry-point to $\mathcal{P}$, then `main` must be safe for all possible inputs. This means that `main` does not require a summary. The conditions are as follows.

$$
\begin{aligned}
\varphi_{Main} := \forall b_1 \cdot \forall b_2 \cdot \forall b_3 \cdot \\
&(b_1 = 1) \Rightarrow \Big(\forall m \cdot Counter_{pre}(m) \wedge \forall m' \cdot \forall p \cdot (Counter_{sum}(m, m', p) \Rightarrow CInv(m', p)\Big) \wedge \\
&(b_1 \neq 1 \wedge b_2 = 1) \Rightarrow (\forall m \cdot \forall p \cdot CInv(m, p) \Rightarrow \\
&\qquad \Big(reset_{pre}(m, p) \wedge \forall m' \cdot \forall p' \Big(reset_{sum}(p, m, p', m') \Rightarrow CInv(p', m')\Big)\Big)) \wedge \\
&(b_1 \neq 1 \wedge b_2 \neq 1 \wedge b_3 = 1) \Rightarrow (\forall m \cdot \forall p \cdot CInv(m, p) \Rightarrow \\
&\qquad \Big(increment_{pre}(m, p) \wedge \forall m' \cdot \forall p' \Big(increment_{sum}(p, m, p', m') \Rightarrow CInv(p', m')\Big)\Big)) \wedge \\
&(b_1 \neq 1 \wedge b_2 \neq 1 \neq b_3 = 1) \Rightarrow (\forall m \cdot \forall p \cdot CInv(m, p) \Rightarrow \\
&\qquad \Big(drain_{pre}(m, p) \wedge \forall m' \cdot \forall p' \Big(drain_{sum}(p, m, p', m') \Rightarrow \top\Big)\Big))
\end{aligned}
$$

As outlined in Sec. 3.3, $\mathsf{ToCHC}(\mathcal{P}) = \varphi_{Main} \wedge \varphi_{Ctor} \wedge \varphi_{Reset} \wedge \varphi_{Cap} \wedge \varphi_{Incr} \wedge \varphi_{Dr}$. Next, $\mathsf{ToCHC}(\mathcal{P})$ is strengthened by the predicate template $\Pi_0(\texttt{CInv})$ to obtain $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0) = \mathsf{ToCHC}(\mathcal{P}) \wedge (\forall m \cdot \forall p \cdot \bot \Rightarrow \texttt{CInv}(m, p))$. Clearly the term $\bot \Rightarrow \texttt{CInv}(m, p)$ is trivially satisfied. This is because the predicate template $\Pi_0(\texttt{CInv})$ is also trivial. In general, this need not be the case. Nonetheless, the term $\mathsf{ToCHC}(\mathcal{P})$ is non-trivial. If $\mathsf{ToCHC}(\mathcal{P})$ is provided to a CHC-solver, then the CHC-solver will return a solution σ containing the following components: expressions $\sigma(Counter_{pre})$, $\sigma(Reset_{pre})$, $\sigma(Capacity_{pre})$, $\sigma(Increment_{pre})$, and $\sigma(Drain_{pre})$, which over-approximate the inputs passed to each function; expressions $\sigma(Counter_{sum})$, $\sigma(Reset_{sum})$, $\sigma(Capacity_{sum})$, $\sigma(Increment_{sum})$, and $\sigma(Drain_{sum})$, which over-approximate the return values of each function; an expression $\sigma(loop_{13})$ which over-approximates the reachable states of the loop in `drain`; an expression $\sigma(\texttt{CInv})$ which describes a safe implementation for `CInv`. In one solution, $\sigma(loop_{13}) = (p \leq m \wedge (x \neq 0 \Rightarrow 0 < p) \wedge (x = 0 \Rightarrow 0 = p))$. This states that the counter is always in a valid position, and in position zero if and only if the capacity returns to zero. In such a solution, it is also possible that $\sigma(\texttt{CInv}) = (m > 0 \wedge p \leq m)$. Clearly $\sigma(\texttt{CInv})$ is an $\mathcal{F}$-solution since $\sigma(\texttt{CInv})$ is a conjunction of linear inequalities. Then by Thm. 10, $\Pi : \texttt{CInv} \rightarrow (m > 0 \wedge p \leq m)$ is a solution to $(\mathcal{P}, \mathcal{T}, \Pi_0)$ with $\mathcal{P}[\Pi]$ both closed and safe. ◀

Like CHC-solving, the general IPS-MP problem is also undecidable. This is because program verification reduces to IPS-MP. Intuitively, if a closed program $\mathcal{P}$ is given to an IPS-MP solver, then a solution to the IPS-MP problem implies that $\mathcal{P}$ is correct, and a witness to unrealizability implies that $\mathcal{P}$ is incorrect. In this way, the halting problem also reduces to IPS-MP.

We show that IPS-MP is undecidable for linear integer arithmetic by reducing the halting problem for 2-counter machines to IPS-MP. Recall that a 2-counter machine is a program with a program counter and two integer variables [49]. The program has a finite number of locations, each with one of four instructions: (1) `inc(x)` increases the variable `x` by 1 and increment the program counter; (2) `dec(x)` decreases the variable `x` by 1 and increment the program counter; (3) `jump(x, i)` goes to location `i` if `x` is 0, else increments the program counter; (4) `halt()` halts execution of the program. The halting program for 2-counter machines is known to be undecidable [49].

▶ **Theorem 14.** *The IPS-MP problem is undecidable for linear integer arithmetic.*

6 From Verification to Synthesis

This section establishes reductions of Sec. 2. Class invariant inference is proven directly. Array abstraction and symmetric ring verification are subsumed by a reduction from parameterized compositional model checking to IPS-MP. Loop invariant synthesis is proven in Appx. C.

```
1 class Cls {
2   int x; int y;
3   Cls(int a) { ... }
4   void f(int a) { ... }
5   void g(int a, int b) { ... }}
6 void func(Cls ob, int a) { ... }
```

(a) The input program.

```
1  bool PRED_TEMPLATE Inv(int x, int y) {
2    return synth(x, y); }
3  void main(int br, int a, int b) {
4    if (br == 0) {
5      Cls ob = Cls(a);
6      assert(Inv(ob));
7    } else if (br == 1) {
8      Cls ob = *; assume(Inv(ob));
9      ob.f(a);
10     assert(Inv(ob));
11   } else if (br==2) {
12     Cls ob = *; assume(Inv(ob));
13     ob.g(a, b);
14     assert(Inv(ob));
15   } else if (br==3) {
16     Cls ob = *; assume(Inv(ob));
17     func(ob, a); }}
```

(b) The IPS-MP reduction.

Figure 10 A reduction from class invariant inference to IPS-MP.

We write Σ for a first-order signature, $\mathcal{V}$ for a set of variables, and $\Pi_\perp$ for a collection of predicate templates which maps each predicate to $\perp$.

6.1 Class Invariant Inference

A *safe class invariant* is a predicate that is true of a class instance after initialization, closed under the execution of each impure class method, and sufficient to prove the correctness of a function taking class instances as arguments [37]. *Class invariant inference* asks to find a safe class invariant given a program. The inference problem is *intensional* if solutions are in the same logical fragment as assertions in the programming language [51]. A definition of (intensional) class invariant inference is found in Def. 15. In this definition, $\mathsf{ToCHC}(f)$ relates the class invariant φ to a summary of each method f in $\mathcal{P}$, and f_{pre} is used to enforce that f is summarized. For simplicity, a class has two fields and two impure methods, each taking at most two arguments (Figure 10a). A generalization to m methods is not difficult. A generalization to n arguments follows immediately.

▶ **Definition 15.** *A* class invariant inference problem *is a tuple* $(\mathcal{P}, \mathcal{T})$ *such that* $\mathcal{P} \in$ *Progs*$(\Sigma, \mathcal{V})$ *is an open program as in Figure 10a and* $\mathcal{T}$ *is a theory. A solution to* $(\mathcal{P}, \mathcal{T})$ *is a* $\varphi \in \mathrm{QFFml}(\Sigma, \{\mathsf{x}, \mathsf{y}\})$ *such that the following are* $\mathcal{T}$*-satisfiable:*

$$\psi_{Init} := \forall V \left(\mathtt{Cls}_{pre}(a) \wedge \mathtt{Cls}_{sum}(a, x, y) \Rightarrow \varphi\right) \qquad \psi_{Close1} := \forall V \left(\varphi \wedge \mathsf{f}_{sum}(x, y, a, x', y') \Rightarrow \varphi'\right)$$

$$\psi_{Close2} := \forall V \left(\varphi \wedge \mathsf{g}_{sum}(x, y, a, b, x', y') \Rightarrow \varphi'\right) \qquad \psi_{Suffic} := \forall V \left(\varphi \Rightarrow \mathtt{func}_{sum}(x, y, a)\right)$$

$$\psi_{Sum} := \mathsf{ToCHC}(\mathcal{P}) \wedge \forall V (\varphi \Rightarrow \mathsf{f}_{pre}(x, y, a) \wedge \mathsf{g}_{pre}(x, y, a, b) \wedge \mathtt{func}_{pre}(x, y, a))$$

▶ **Theorem 16.** *Let* $(\mathcal{P}, \mathcal{T})$ *be a class invariant inference problem and* $\mathcal{P}'$ *be the program obtained by adding* `main` *in Figure 10b to* $\mathcal{P}$*. Then* Π *is a solution to* $(\mathcal{P}', \Pi_\perp, \mathcal{T})$ *if and only if* $\Pi(\mathtt{Inv})$ *is a solution to* $(\mathcal{P}, \mathcal{T})$*.*

6.2 Reducing PCMC to IPS-MP

Parameterized compositional model checking (PCMC) is a framework to verify structures with arbitrarily many components (e.g., an array with arbitrarily many cells, or a ring with arbitrarily many processes) by decomposing the structure into smaller structures of fixed sizes [50]. Intuitively, each of these smaller structures is a view of the larger structure from

```
struct View { int l; int r; int s; };
// Transition relation.
View tr(View v) { ... }
// Initial state predicate.
bool init(int l, int s, int r) { ... }
// Correctness property.
bool property(View v) { ... }
```

(a) The input program.

```
bool PRED_TEMPLATE Inv(
  int l, int s, int, r) {
  if (init(l, s, r)) { return true; }
  else { return synth(l, s, r); } }
void main(int br, struct View v) {
  if (br == 0) {
    int inf = *;
    assume(Inv(v.left, v.st, v.right));
    assume(Inv(v.right, inf, v.left));
    v = tr(v);
    assert(Inv(v.left, v.st, v.right));
    assert(Inv(v.right, inf, v.left));
  } else if (br == 1) {
    assume(Inv(v.left, v.st, v.right));
    assert(property(v)); }}
```

(b) The IPS-MP reduction.

Figure 11 A reduction from compositional ring invariant synthesis to IPS-MP. The state of each process and resource are both assumed to be integer values.

the perspective of a single component. A proof of the larger structure is obtained by verifying each of the smaller structures, and showing that their proofs compose with one another [50]. If the number of smaller structures is finite (i.e., most perspectives are similar), then PCMC is applicable [50]. For example, in Sec. 2.3 and Sec. 2.4, the array and ring were highly symmetric, and therefore, all perspectives were similar.

Once the larger structure has been decomposed, the proof of compositionality follows by inferring *adequate compositional invariants* for groups of similar components [50]. The number of compositional invariants, and the properties they must satisfy, depend on the decomposition. However, each property is one of initialization, closure, or non-interference. An initialization property states that a compositional invariant is true for the initial state of a component. A closure property states that a compositional invariant is closed under all transitions of its components. A non-interference property states that for any component c, if c satisfies its compositional invariant and an adjacent component (*also satisfying its compositional invariant*) performs a transition, then c continues to satisfy its compositional invariant after the transition. In addition, all composition invariants must be adequate in that they imply the correctness of the larger structure. To make the rest of this section concrete, we restrict ourselves to compositional ring invariants[6]. As in Sec. 6.1, the inference problem is assumed to be intensional. A formal definition of (intensional) compositional invariant inference is given in Def. 17[7]. Note that in Def. 17 ToCHC relates the compositional invariant φ to the summary of $\mathtt{tr}$, $\mathtt{tr}_{pre}$ enforces that $\mathtt{tr}$ is summarized, and $\varphi_{Inf} := \varphi[l/r][s/i][r/l]$ is the compositional invariant applied to a process (r, i, l).

▶ **Definition 17.** *A* compositional ring invariant (CRI) inference problem *is a tuple* $(\mathcal{P}, \mathcal{T})$ *such that* $\mathcal{P} \in \mathit{Progs}(\Sigma, \mathcal{V})$ *is an open program as in Figure 11a and* $\mathcal{T}$ *is a theory. A solution to* $(\mathcal{P}, \mathcal{T})$ *is a* $\varphi \in \mathrm{QFFml}(\Sigma, \{\mathtt{l}, \mathtt{s}, \mathtt{r}\})$ *such that the following are* $\mathcal{T}$*-satisfiable given* $\varphi_{Inf} := \varphi[l/r][s/i][r/l]$*:*

$$\psi_{Init} := \forall V \left(\mathtt{init}(l, s, r) \Rightarrow \varphi\right) \qquad \psi_{Close} := \forall V \left(\varphi \wedge \varphi_{Inf} \wedge \mathtt{tr}_{sum}(l, s, r, l', s', r') \Rightarrow \varphi'\right)$$

$$\psi_{Adeq} := \forall V \left(\varphi \Rightarrow \mathtt{property}(l, s, r)\right) \qquad \psi_{Inf} := \forall V \left(\varphi \wedge \varphi_{Inf} \wedge \mathtt{tr}_{sum}(l, s, r, l', s', r') \Rightarrow \varphi_{Inf}'\right)$$

$$\psi_{Sum} := \mathsf{ToCHC}(\mathcal{P}) \wedge \forall V \cdot (\varphi \wedge \varphi_{Inf} \Rightarrow \mathtt{tr}_{pre}(l, s, r))$$

[6] Sec. 2.3 is a degenerate case. In this ring, processes communicate through locks. In an array, cells do not "communicate".

[7] In PCMC, a witness to unrealizability does not entail the incorrectness of a structure. Instead, no proof of correctness exists relative to the chosen decomposition.

Benchmarks					IPS-MP (Spacer)			IPS-MP (Eldarica)				HornSpec			CVC4		
Type	Safe	Buggy	Preds	Size	Time	TO	✓	Time	TO	MEM	✓	Time	UN	✓	TO	N/A	✓
Loop	7	7	9	179 KB	**4**	0	**14**	45	0	0	**14**	4	12	2	7	7	0
Class	6	6	6	694 KB	**2**	0	**12**	1 449	0	0	**12**	—	12	0	6	6	0
Array	4	6	6	535 KB	**4**	0	**10**	230	0	0	**10**	—	10	0	4	6	0
Ring	2	3	2	197 KB	**1**	0	**5**	52	0	0	**5**	—	5	0	2	3	0
Proc	3	3	3	418 KB	**2**	0	**6**	4	0	0	**6**	—	6	0	3	3	0
SC	70	4	181	974 MB	**6 878**	4	**70**	6 717	53	12	9	—	—	—	—	—	—
Total	92	29	207	975 MB	**6 891**	4	**117**	8 497	53	12	56	4	45	2	22	29	0

Table 1 Performance of various solvers on IPS-MP benchmarks.

▶ **Theorem 18.** *Let $(\mathcal{P}, \mathcal{T})$ be a CRI inference problem, $\mathcal{P}'$ be the program obtained by adding* `main` *of Figure 11b to $\mathcal{P}$, and Π_0 be the predicate template from Figure 11b. Then Π is a solution to $(\mathcal{P}', \Pi_0, \mathcal{T})$ if and only if $\Pi(\texttt{Inv})$ is a solution to $(\mathcal{P}, \mathcal{T})$.*

7 Implementation and Evaluation

We have implemented an IPS-MP solver within the SeaHorn verification framework. SeaHorn takes as input a C program, and returns a CHC-based verification problem in the SMT-LIB format according to Sec. 3.3 [32]. We extend SeaHorn to recognize predicate templates. For each predicate, SeaHorn adds clauses to the verification conditions according to Sec. 5.2. Proofs of unrealizability are generated with the implementation of [28] found in SeaHorn. That is, proofs of unrealizability are already supported by SeaHorn.

The goal of our evaluation is to confirm that:

1. IPS-MP is practical for the reduction described in Sec. 6;
2. CHC-based solvers are more efficient than general synthesis solvers for IPS-MP instances;
3. The overhead incurred when using IPS-MP is tolerable.

Towards (1) and (2), we have collected 92 IPS-MP problems with linear integer arithmetic as the background theory (see Safe in Tab. 1). Of these benchmarks, 7 reflect loop invariant inference (and interpolation [48]), 6 reflect class invariant synthesis, 4 reflect array (and memory) abstraction, 2 reflect ring PCMC, 3 reflect procedure summarization, and 70 reflect parameterized analysis of smart-contract (SC) programs (see [67, 68]). The first 20 benchmarks were collected from research papers in the area of software verification. The remaining benchmarks, involving the parameterized analysis of SCs, were obtained by extending SmartACE with support for IPS-MP. Of these 70 SC benchmarks, 62 are taken from real-world examples used to manage monetary assets [53]. To address question (3), we compare the performance of SmartACE with and without IPS-MP, relative to these real-world examples. Note that the extension to SmartACE was a routine exercise, due to the original design of SmartACE. In particular, SmartACE encodes all compositional invariants as predicates returning `true`, to then be refined manually by an end-user [68]. These predicates appear in assume and assert statements, as described in Sec. 2.4. Our extended version of SmartACE can replace these predicates with predicate templates, yielding valid IPS-MP problems.

A summary of all benchmarks can be found in Tab. 1. As reflected by their size (see Size in Tab. 1) and total number of unknown predicates across all realizable instances (see Preds in Tab. 1), SCs are included to evaluate IPS-MP on large programs. When possible, benchmarks are drawn from prior works in program analysis (i.e., [40, 46, 59, 53]). To reflect unrealizability in IPS-MP, 29 faults have been injected in these benchmarks (see Buggy in Tab. 1). Further information can be found about the realizable real-world SC's in Tab. 2.

Contracts					Performance (Time)		
Name	Props	Preds	Size	✓	VerX [53]	SmartACE (Manual) [67]	SmartACE (IPS-MP)
Alchemist	3	12	36 MB	3	29	7	208
Brickblock	6	12	122 MB	6	191	13	1214
Crowdsale	9	27	76 MB	9	261	223	238
ERC20	9	27	45 MB	9	158	12	103
Melon	16	32	149 MB	16	408	30	979
PolicyPal	4	16	123 MB	4	20 773	26	3118
VUToken	5	22	319 MB	1	715	19	17
Zebi	5	14	45 MB	5	77	8	487
Zilliqa	5	10	54 MB	5	94	8	501
Total	62	172	969 MB	58	22 706	346	5685

Table 2 Overhead of integrating IPS-MP-solving with SmartACE.

Each SC in this table is associated with one or more safety properties (see Props in Tab. 2), which in turn, corresponds to a realizable IPS-MP instance. As before, Preds and Size indicate the total predicate count and size for these instances. All benchmarks are available at `https://doi.org/10.5281/zenodo.5083785`.

To evaluate IPS-MP, we find the number of benchmarks that are solved by either of two state-of-the-art CHC solvers: Eldarica [35] and Spacer [43]. To compare CHC solvers to general synthesis tools, we provide our benchmarks to a state-of-the-art specification synthesizer, HornSpec [54], and a state-of-the-art SyGuS solver, CVC4[8] [11]. Since CVC4 solves SyGuS instances, which do not support proofs on unrealizability, then we only evaluate CVC4 on realizable benchmarks (see N/A in Tab. 1). Due to the size of each SC benchmark, we only ran the tools that could solve Loop through to Proc on these benchmarks. The results for each tool are reported in Tab. 1, where TO is the number of timeouts (after 30 minutes), MEM is the number of failures due to memory limits, UN is the number of benchmarks for which a tool returned `unknown`, ✓ is the number of benchmarks solved, and Time is the total time (in seconds) to find all solutions in a given set. In Tab. 2, the total time for Spacer is further broken down by SC (see SmartACE (IPS-MP) in Tab. 2). For comparison, the verification times for VerX (an automated SC verifier with user-guided predicate abstraction [53]) and the original version of SmartACE (see SmartACE (Manual) in Tab. 2) are also provided. All evaluations were run on an Intel® Core i7® CPU @ 1.8GHz 8-core machine with 16GB of RAM running Ubuntu 20.04.

From this evaluation, we answer questions (1) through to (3) in the positive.

1. As illustrated by Tab. 1, many examples of class invariant inference and compositional invariant inference (i.e., Class, Array, Ring, and SC) taken from the literature could be encoded using IPS-MP. In the case of SC, the generation of IPS-MP instances could be fully-automated using a modified version of SmartACE. We conclude that IPS-MP is practical for the reductions described in Sec. 6.
2. As shown in Tab. 1, all small benchmarks were solved by Eldarica and Spacer, with average times under a minute. Furthermore, all but four SC benchmarks were solved by Spacer within a 30-minute timeout, with an average time of 96 seconds. Upon closer inspection, we found that Spacer would fail to solve these four examples, and would return `unknown` after approximately one hour. However, CVC4 failed to solve any SC benchmarks within 30-minutes. Therefore, we conclude that CHC-based IPS-MP-solving is effective for the reductions of Sec. 6, and can outperform general synthesis solutions.

[8] To support CVC4, we convert each *realizable* problem from SMT-LIB format to the SyGuS input language.

We note that HornSpec returned `unknown` on all but two benchmarks[9].

3. As shown in Tab. 2, the IPS-MP version of SmartACE incurred an average time overhead of 18x as compared to the manual version of SmartACE. This should come as no surprise, since the manual version of SmartACE achieved a verification time of under 3 seconds for 44 of the 62 properties with the help of user-provided compositional invariants. In these cases, a solving time as low as 60 seconds would correspond to an overhead of at least 20x. To better contextualize this overhead, we compare the verification time of IPS-MP version of SmartACE to the verification time of VerX. We first note the outlying case of PolicyPal, in which the IPS-MP version of SmartACE achieves a speedup of over 6x. For the remaining SC's, the IPS-MP version of SmartACE fell within 1.3x of VerX on average. Since VerX is a specialized tool with less automation than the IPS-MP version of SmartACE, we conclude that the overhead incurred by IPS-MP is tolerable in this particular real-world application. We note that in [53], only the "average" times were reported for VerX. It is unclear whether this is the average time to verify all properties, or an average across all properties. The authors of VerX were contacted, but were unable to provide the original data. For this reason, we assume conservatively that all times reported by VerX are total.

One limitation of the evaluation is its emphasis on SC verification. However, compositional SC verification is representative of compositional verification, as illustrated in [67]. We do acknowledge that design patterns specific to SC development might bias the benchmark set. We hope for this benchmark set to be expanded in future work.

Note, however, that we do not plan to benchmark our IPS-MP solver against invariant synthesis tools. Recall that our implementation simply extends SeaHorn with support for the IPS-MP synthesis language. In cases where the IPS-MP instance reduces to invariant synthesis, our extension is bypassed, and verification reduces to executing SeaHorn. Therefore, a direct comparison is not possible, and the evaluation results would not be meaningful. Furthermore, SeaHorn is a state-of-the-art program verifier with prior success in SV-COMP. Thus, SeaHorn is already known to perform well on invariant synthesis tasks.

An important direction for future work is to understand why CVC4 times out on all benchmarks. We hypothesize that the lack of a grammar in IPS-MP proves challenging for CVC4's enumerative search. We also note that many of our benchmarks produce non-linear CHC's, whereas the invariant synthesis track for SyGuS reduces to solving linear CHC's.

8 Related Work

General program synthesis. As explained in Sec. 1, general synthesis engines (e.g., Sketch [62], Rosette [64], SyGuS [5], and SemGuS [42]) are fundamentally different from IPS-MP. Among these frameworks, only SemGuS can both solve synthesis problems and prove unrealizability. Similar to IPS-MP, SemGuS reduces the synthesis problem to satisfiability of CHCs. However, this is where the similarities end. SemGuS reduces synthesis to unsatisfiability and extracts solutions from the refutation proofs. In contrast, IPS-MP reduces to satisfiability and solutions are extracted from model of the CHCs. SemGuS solves a more general problem, which comes at a high price both from a theoretical and practical perspective. We show that IPS-MP modulo Boolean programs can be solved in polynomial time (in the number of states), while SemGuS lacks this guarantee. Existing SemGuS solvers (e.g., Messy [42]) synthesize programs from sets of candidates described

[9] The authors of HornSpec confirm this result though the cause is unknown.

using regular tree grammars. As a result, their CHCs use constraints over Algebraic Data Types to represent the grammar terms, which are harder to solve than either Boolean or linear arithmetic constraints. Only Sketch and Rosette are "modulo programs", but do not allow loops nor recursion.

Specification synthesis. Specification synthesis solves the problem of finding specifications for unknown procedures which enable the verification of a given program (e.g., [21, 2, 54]). Unlike IPS-MP, specification synthesis is under-specified. Trivial specifications such as *false* are often sufficient but undesirable. As a consequence, many tools aim to synthesize either *weakest* (i.e., maximal) or non-vacuous solutions. In IPS-MP, any solution is valid as long as it satisfies all program assertions. In Sec. 7, we also compare our IPS-MP solver with HornSpec [54] and demonstrate that HornSpec is unsuitable for IPS-MP.

Data-driven invariant generation. Multiple approaches have been proposed (e.g., [27, 71, 60, 69, 58, 36]) that rephrase loop invariant synthesis as a learning problem. Recent work has extended these techniques to parameterized verification [70]. Often, these techniques require problem-specific biases to learn useful invariants (e.g., [60, 69, 58, 70]). Furthermore, these techniques lack the complexity bounds of decidable verification. In contrast, IPS-MP is problem-agnostic, and achieves the same complexity as verification in the Boolean case. Adapting data-driven techniques to IPS-MP-solving is an interesting future direction.

Constrained Horn clauses. In recent years, CHC-solvers have become a common tool for verification and synthesis problems. Example include SeaHorn [32], SemGuS [42], and HornSpec [54]. The connection between CHCs and verification has long been explored in the CLP community (e.g., [39, 52, 22]). This direction was popularized again by the work of Rybalchenko et al. [30]. According to the annual CHC-COMP competition[10], Spacer [43] and Eldarica [35] are the most effective general-purpose CHC-solvers.

9 Conclusion

We proposed IPS-MP, a novel synthesis problem suitable for solving a wide range of verification problems, such as invariant inference and verification of parameterized systems. To demonstrate the relevance of IPS-MP, we provided three reductions from classic verification problems to IPS-MP. To highlight IPS-MP's practicality, we proposed a solution that effectively leverages off-the-shelf CHC solvers and implemented it in the SeaHorn verification framework. Our evaluation demonstrates the effectiveness of CHC solvers in solving IPS-MP when compared with general synthesis tools such as HornSpec and CVC4.

Finally, we demonstrated that the interesting instance of IPS-MP for Boolean programs is efficiently decidable, whereas the general instance is undecidable. Despite this, the general instance of IPS-MP is theoretically simpler than general synthesis, and thus, warrants specialized solvers. In future work, we plan to study other instances of IPS-MP, such as IPS modulo timed automata. We further suspect that IPS-MP will enable new practical applications of PCMC.

References

1 Aneesh Aggarwal and Keith H. Randall. Related field analysis. In *PLDI*, pages 214–220. ACM, 2001.

2 Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *POPL*, pages 789–801. ACM, 2016.

[10] https://chc-comp.github.io.

3 Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, volume 7358 of *LNCS*, pages 672–678. Springer Berlin Heidelberg, 2012.

4 Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. SAFEVM: a safety verifier for Ethereum smart contracts. In *ISSTA*, pages 386–389. ACM, 2019.

5 Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015.

6 Rajeev Alur, Ahmed Bouajjani, and Javier Esparza. Model checking procedural programs. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 541–572. Springer Cham, 2018.

7 Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. SyGuS-Comp 2016: Results and analysis. In *SYNT@CAV*, volume 229 of *EPTCS*, pages 178–202, 2016.

8 Thomas Ball and Sriram K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE*, pages 97–103. ACM, 2001.

9 Jiri Barnat, Lubos Brim, Milan Češka, and Petr Ročkai. DiVinE: Parallel distributed model checker. In *PDMC–HIBI*, pages 4–7. IEEE, 2010.

10 Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer Berlin Heidelberg, 2005.

11 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer Berlin Heidelberg, 2011.

12 Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer Cham, 2018.

13 Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. Building code transpilers for domain-specific languages using program synthesis. In *ECOOP*, volume 263 of *LIPIcs*, pages 38:1–38:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

14 Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, volume 9300 of *LNCS*, pages 24–51. Springer Berlin Heidelberg, 2015.

15 Roderick Bloem, Swen Jacobs, and Yakir Vizel. Efficient information-flow verification under speculative execution. In *ATVA*, pages 499–514. Springer Cham, 2019.

16 James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with Metasketches. In *POPL*, pages 775–788. ACM, 2016.

17 Sagar Chaki and Arie Gurfinkel. BDD-based symbolic model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 219–245. Springer Cham, 2018.

18 Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable and efficient secure two-party computation for machine learning. In *EuroS&P*, pages 496–511. IEEE, 2019.

19 K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.

20 Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118. ACM, 2011.

21 Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *CAV*, volume 9206 of *LNCS*, pages 324–342. Springer Berlin Heidelberg, 2015.

22 Giorgio Delzanno and Andreas Podelski. Model checking in CLP. In *TACAS*, volume 1579 of *LNCS*, pages 223–239. Springer Berlin Heidelberg, 1999.

23 Edsger Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453—-457, 1975.

24 Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, USA, 2002.

25 Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. Sampling invariants from frequency distributions. In *FMCAD*, pages 100–107. IEEE, 2017.

26 Jean-Christophe Filliâtre and Andrei Paskevich. Why3—Where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer Berlin Heidelberg, 2013.

27 Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *CAV*, volume 8559 of *LNCS*, pages 69–87. Springer Berlin Heidelberg, 2014.

28 Jeffrey Gennari, Arie Gurfinkel, Temesghen Kahsai, Jorge A. Navas, and Edward J. Schwartz. Executable counterexamples in software model checking. In *VSTTE*, volume 11294 of *LNCS*, pages 17–37. Springer, 2018.

29 Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350. ACM, 2005.

30 Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.

31 David Gries. A note on a standard strategy for developing loop invariants and loops. *Sci. Comput. Program.*, 2(3):207–214, 1982.

32 Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *CAV*, volume 9206 of *LNCS*, pages 343–361. Springer Berlin Heidelberg, 2015.

33 Arie Gurfinkel and Jorge A. Navas. Abstract interpretation of LLVM with a region-based memory model. In Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina, editors, *Software Verification*, volume 13124 of *LNCS*, pages 122–144. Springer, 2022.

34 Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

35 Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn solver. In *FMCAD*, pages 1–7. IEEE, 2018.

36 Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *PLDI*, pages 1159–1174. ACM, 2020.

37 Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In *FASE*, volume 1783 of *LNCS*, pages 208–221. Springer Berlin Heidelberg, 2000.

38 Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL*, pages 111–119. ACM, 1987.

39 Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. A CLP proof method for timed automata. In *RTSS*, pages 175–186. IEEE Computer Society, 2004.

40 Temesghen Kahsai, Rody Kersten, Philipp Rümmer, and Martin Schäf. Quantified heap invariants for object-oriented programs. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 368–384. EasyChair, 2017.

41 Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *NDSS*. The Internet Society, 2018.

42 Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas W. Reps. Semantics-Guided Synthesis. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021.

43 Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34. Springer Berlin Heidelberg, 2014.

44 Jakub Kuderski, Jorge A. Navas, and Arie Gurfinkel. Unification-based pointer analysis without oversharing. In *FMCAD*, pages 37–45. IEEE, 2019.

45 Jérôme Leroux, Philipp Rümmer, and Pavle Subotic. Guiding Craig interpolation with domain-specific abstractions. *Acta Informatica*, 53(4):387–424, 2016.

46 Francesco Logozzo. Automatic inference of class invariants. In *VMCAI*, volume 2937 of *LNCS*, pages 211–222. Springer Berlin Heidelberg, 2004.

47 Sirui Lu and Rastislav Bodík. Grisette: Symbolic compilation as a functional programming library. *Proc. ACM Program. Lang.*, 7(POPL), 2023.

48 Kenneth McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer Berlin Heidelberg, 2003.

49 Marvin Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

50 Kedar S. Namjoshi and Richard J. Trefler. Parameterized compositional model checking. In *TACAS*, volume 9636 of *LNCS*, pages 589–606. Springer Berlin Heidelberg, 2016.

51 Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007.

52 Julio C. Peralta, John P. Gallagher, and Hüseyin Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *SAS*, volume 1503 of *LNCS*, pages 246–261. Springer Berlin Heidelberg, 1998.

53 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. VerX: Safety verification of smart contracts. In *S&P*, pages 1661–1677. IEEE, 2020.

54 Sumanth Prabhu, Grigory Fedyukovich, Kumar Madhukar, and Deepak D'Souza. Specification synthesis with constrained Horn clauses. In *PLDI*, pages 1203–1217. ACM, 2021.

55 Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV*, volume 8559 of *LNCS*, pages 106–113. Springer Cham, 2014.

56 Dan Rasin, Orna Grumberg, and Sharon Shoham. Modular verification of concurrent programs via sequential model checking. In *ATVA*, pages 228–247. Springer Cham, 2018.

57 Heinz Riener and Görschwin Fey. FAuST: A framework for formal verification, automated debugging, and software test generation. In *SPIN*, volume 13872 of *LNCS*, pages 234–240. Springer Berlin Heidelberg, 2012.

58 Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: learning loop invariants with continuous logic networks. In *ICLR*. OpenReview.net, 2020.

59 Malte Hermann Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, Switzerland, 2016.

60 Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2Inv: A deep learning framework for program verification. In *CAV*, volume 12225 of *LNCS*, pages 151–164. Springer Berlin Heidelberg, 2020.

61 Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In *SSV*, page 7, USA, 2010. USENIX Association.

62 Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013.

63 Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *PLDI*, pages 136–148. ACM, 2008.

64 Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, pages 530–541. ACM, 2014.

65 Moshe Y. Vardi. From verification to synthesis. In *VSTTE*, volume 5295 of *LNCS*, page 2. Springer Berlin Heidelberg, 2008.

66 Hari Govind VK, Yuting Chen, Sharon Shoham, and Arie Gurfinkel. Global guidance for local generalization in model checking. In Shuvendu K. Lahiri and Chao Wang, editors, *CAV*, volume 12225 of *LNCS*, pages 101–125. Springer Berlin Heidelberg, 2020.

67 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholz, and Arie Gurfinkel. Compositional verification of smart contracts through communication abstraction. In *Static Analysis*, volume 12913 of *LNCS*, pages 429–452. Springer Berlin Heidelberg, 2021.

68 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholz, and Arie Gurfinkel. Verifying Solidity smart contracts via communication abstraction in SmartACE. In *VMCAI*, pages 425–449. Springer Cham, 2022.

69 Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *PLDI*, pages 106–120. ACM, 2020.

70 Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-driven automated invariant learning for distributed protocols. In *OSDI*, pages 405–421. USENIX Association, 2021.

71 He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *PLDI*, pages 707–721. ACM, 2018.

$$
\begin{array}{rcl}
\langle\text{Procs}\rangle & ::= & \textit{a procedure name} \mid \texttt{Main} \\
\langle\text{Preds}\rangle & ::= & \textit{a predicate name} \\
\langle\text{VarList}\rangle & ::= & \langle\text{Var}\rangle\texttt{,}\ \ldots\texttt{,}\ \langle\text{Var}\rangle \\
\langle\text{ValList}\rangle & ::= & \langle\text{Expr}\rangle\texttt{,}\ \ldots\texttt{,}\ \langle\text{Expr}\rangle \\
\langle\text{ProcApp}\rangle & ::= & \langle\text{Procs}\rangle\texttt{(}\ \langle\text{ValList}\rangle\ \texttt{)} \\
\langle\text{PredApp}\rangle & ::= & \langle\text{Preds}\rangle\texttt{(}\ \langle\text{ValList}\rangle\ \texttt{)} \\
\langle\text{Inst}\rangle & ::= & \langle\text{Var}\rangle\ \texttt{=}\ \langle\text{Val}\rangle \mid \langle\text{Var}\rangle\ \texttt{=}\ \texttt{*} \mid \texttt{skip} \mid \texttt{assume(}\ \langle\text{Expr}\rangle\ \texttt{)} \mid \\
 & & \texttt{assert(}\ \langle\text{Expr}\rangle\ \texttt{)} \mid \texttt{assume(}\ \langle\text{PredApp}\rangle\ \texttt{)} \mid \\
 & & \texttt{assert(}\ \langle\text{PredApp}\rangle\ \texttt{)} \mid \langle\text{VarList}\rangle\ \texttt{=}\ \langle\text{ProcApp}\rangle \\
\langle\text{Stmt}\rangle & ::= & \langle\text{Stmt}\rangle\texttt{;}\ \langle\text{Stmt}\rangle \mid \langle\text{Inst}\rangle \mid \texttt{while}\ \texttt{(}\ \langle\text{Expr}\rangle\ \texttt{)}\ \texttt{\{}\ \langle\text{Stmt}\rangle\texttt{;}\ \texttt{\}} \mid \\
 & & \texttt{if}\ \texttt{(}\ \langle\text{Expr}\rangle\ \texttt{)}\ \texttt{\{}\ \langle\text{Stmt}\rangle\texttt{;}\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ \langle\text{Stmt}\rangle\texttt{;}\ \texttt{\}} \\
\langle\text{ProcDecl}\rangle & ::= & \langle\text{Procs}\rangle\texttt{(}\ \langle\text{VarList}\rangle\ \texttt{)}\ \texttt{\{}\ \langle\text{Stmt}\rangle\texttt{;}\ \texttt{return}\ \langle\text{ValList}\rangle\texttt{;}\ \texttt{\}} \\
\langle\text{PredDecl}\rangle & ::= & \langle\text{Preds}\rangle\texttt{(}\ \langle\text{VarList}\rangle\ \texttt{)}\ \texttt{\{}\ \texttt{return}\ \langle\text{Expr}\rangle\texttt{;}\ \texttt{\}}
\end{array}
$$

Figure 12 The formal grammar for programs with variables $\mathcal{V}$ and operations over the signature Σ. That is, $\langle\text{Var}\rangle ::= \mathcal{V}$, $\langle\text{Val}\rangle ::= \text{Term}(\Sigma, \mathcal{V})$, and $\langle\text{Expr}\rangle ::= \text{QFFml}(\Sigma, \mathcal{V})$. The set of programs in the language is denoted by $\mathsf{Progs}(\Sigma, \mathcal{V})$.

A Syntax and Semantics

The syntax of $\mathsf{Progs}(\Sigma, \mathcal{V})$ is presented in Figure 12. To simplify the presentation, types are omitted and all local variables are declared as inputs to procedures. Up to these simplifications, all IPS-MP instances in Sec. 2 can be thought of as programs in this language. A qualitative description of this language can be found in Sec. 3.2. Denotational semantics for this language are given by the WLP transformer in Figure 13. In this semantic interpretation, each loop S at line ln is associated with a predicate $loop_{ln}$. The result of $wlp(S, Q)$ encodes that $loop_{ln}$ is a loop invariant for S which is sufficient to entail Q.

B Loop Invariant Inference as Synthesis

Consider the program in Figure 14a. This program takes as input a non-negative integer `x`, and then computes $2 \cdot x$ through repeated addition. The function is correct if $y = 2 \cdot x$ on line 5. A standard approach to this problem is to find an invariant for the loop on line 4 that entails $y = 2 \cdot x$ on line 5. Therefore, the goal of this example is to construct an IPS-MP problem to synthesize such a loop invariant (existence of this loop invariant entails the correctness of Figure 14a).

By definition, a loop invariant is a predicate that is true upon entry to the loop, closed under each iteration of the loop, and true of the program's state upon loop termination [31]. Each requirement of a loop invariant can be represented through assumptions and assertions, as in Figure 14b. First, to ensure that the loop invariant is true upon entry, the loop invariant is asserted upon entering the loop (line 8). Second, to ensure that the loop invariant is closed under each iteration of the loop, Figure 14b over-approximates the state of the program upon entry to an arbitrary iteration of the loop. To restrict the program to an arbitrary iteration of the loop, the loop is first unrolled to a single iteration (lines 11–14). Before

$$
\begin{aligned}
wlp(S_1; S_2, Q) &:= wlp(S_1, wlp(S_2, Q)) \\
wlp(\mathsf{while}_{ln}\ (\varphi)\ \{S\}, Q) &:= \forall \vec{w} \cdot ((loop_{ln}(\vec{w}) \wedge \varphi) \Rightarrow wlp\,(S, loop_{ln}(\vec{w}))) \wedge \\
&\quad\ \forall \vec{w} \cdot ((loop_{ln}(\vec{w}) \wedge \neg\varphi) \Rightarrow Q) \wedge loop_{ln}(\vec{w}) \\
wlp(\mathsf{if}\ (\varphi)\ \{S_1\}\ \mathsf{else}\ \{S_2\}, Q) &:= (\varphi \Rightarrow wlp(S_1, Q)) \wedge (\neg\varphi \Rightarrow wlp(S_2, Q)) \\
wlp(x = *, Q) &:= \forall x \cdot Q \\
wlp(\vec{y} = f(\vec{e}), Q) &:= f_{pre}(\vec{e}) \wedge \forall \vec{r} \cdot (f_{sum}(\vec{e}, \vec{r}) \Rightarrow Q[\vec{y}/\vec{r}]) \\
wlp(\mathsf{skip}, Q) &:= Q \\
wlp(x = e, Q) &:= Q[x/e] \\
wlp(\mathsf{assert}(\varphi), Q) &:= \varphi \wedge Q \\
wlp(\mathsf{assume}(\varphi), Q) &:= \varphi \Rightarrow Q
\end{aligned}
$$

Figure 13 The WLP transformer for Figure 12. This follows the presentation of [14].

```
1 void main(int x) {
2   assume(x >= 0);
3   int i = 0; int y = 0;
4   while (i < x) { i += 1; y += 2;}
5   assert(y == 2 * x); }
```

(a) The original program.

```
1  bool PRED_TEMPLATE Inv1(int x, int i) {
2    return synth(x, i); }
3  bool PRED_TEMPLATE Inv2(int i, int y) {
4    return synth(i, y); }
5  void main(int x) {
6    assume(x >= 0);
7    int i = 0; int y = 0;
8    assert(Inv1(x, i)); assert(Inv2(i, y));
9    y = *; i = *;
10   assume(Inv1(x, i)); assume(Inv2(i, y));
11   if (i < x) { // Unrolled while loop.
12     i += 1; y += 2;
13     assert(Inv1(x, i)); assert(Inv2(i, y));
14     assume(false); }
15   assert(y == 2 * x); }
```

(b) The IPS-MP problem.

Figure 14 A program that is correct relative to the loop invariant $(2 \cdot i = y) \wedge (i \leq x)$, and an IPS-MP problem that synthesizes the loop invariant.

checking the loop condition, the state of an arbitrary loop iteration is then selected by setting each mutable variable non-deterministically, and assuming that these new values satisfy the loop invariant (lines 9–10). If this state also satisfies the loop condition, then closure is enforced by first executing the body of the loop, and then asserting that the loop invariant is maintained (lines 12–14). Otherwise, the state does not satisfy the loop condition, and the program exits the loop (line 15). This gives a program with unknowns, as required by the verification methodology.

Next, the shape of the loop invariant is restricted. We note that the loop is correct because `i` is incremented up to `x`, while maintaining that `y` is twice the value of `i`. It follows that there exists a loop invariant for Figure 14a that does not relate `x` and `y` in the same constraint. However, `x` must be compared with `i`, and `i` must be compared with `y`. Therefore, our loop invariant has the shape $\texttt{Inv1}(x, i) \wedge \texttt{Inv2}(i, y)$. In the IPS-MP encoding, both `Inv1` and `Inv2` correspond to partial predicates (see lines 1 and 3, respectively) that are assumed and asserted together (see lines 10 and 8, respectively). One solution to this problem is to assign the expression `(i <= x)` to the hole in `Inv1`, and the expression `(y == 2 * i)` to the hole in `Inv2`.

C Loop Invariant Inference: Reduction to IPS-MP

A *safe loop invariant* is a predicate that is true upon entry to a loop, maintained by each iteration of a loop, and is sufficient to prove that the program is correct [31]. *Loop invariant inference* asks to find a safe loop invariant given a program. The inference problem is *intensional* if solutions must be in the same logical fragment as assertions in the programming language [51]. A formal definition of (intensional) safe loop invariant inference is given in Def. 19. Note that in Def. 19, $\mathsf{ToCHC}(f)$ is used to relate the loop invariant φ to the summary of each procedure f in $\mathcal{P}$. For simplicity of presentation, a program has a single loop and two variables (Figure 15a). A generalization to m loops is possible and not difficult. A generalization to n variables follows immediately.

▶ **Definition 19.** *A* loop invariant inference problem *is a tuple* $(\mathcal{P}, \mathcal{T})$ *such that* $\mathcal{P} \in \mathsf{Progs}(\Sigma, \{x_1, x_2\})$ *is a problem with the* `main` *procedure from Figure 15a (where* Σ *is a first-order signature), and* $\mathcal{T}$ *is a theory. A solution to* $(\mathcal{P}, \mathcal{T})$ *is a* $\varphi \in \mathrm{QFFml}(\Sigma, \{x_1, x_2\})$ *such that the following are* $\mathcal{T}$*-satisfiable:*

1. $\psi_{Pre} := \forall V \cdot wlp(S_1, \varphi)$*, where* S_1 *is the statement before the loop;*
2. $\psi_{Tr} := \forall V \cdot (\varphi \wedge e) \Rightarrow wlp(S_2, \varphi)$*, where* S_2 *is the loop body and* e *is the loop condition;*
3. $\psi_{Post} := \forall V \cdot (\varphi \wedge \neg e) \Rightarrow wlp(S_3, \top)$*, where* S_3 *is the statement after the loop;*
4. $\psi_{Procs} := \bigwedge_{f \in \mathsf{Procs}(\mathcal{P})} \mathsf{ToCHC}(f)$.

▶ **Theorem 20.** *Let* $(\mathcal{P}, \mathcal{T})$ *be a loop invariant inference problem and* $\mathcal{P}'$ *be the program obtained by replacing* `main` *in* $\mathcal{P}$ *with the definition of* `main` *in Figure 15b. The partial predicate implementation* Π *is a solution to* $(\mathcal{P}', \Pi_\perp, \mathcal{T})$ *if and only if* $\Pi(\mathtt{Inv})$ *is a solution to* $(\mathcal{P}, \mathcal{T})$.

Proof. Let Π be a solution $(\mathcal{P}', \Pi_\perp, \mathcal{T})$. This is true if and only if $\mathcal{P}'[\Pi]$ is correct relative to $\mathcal{T}$. By Prop. 22, this is true if and only if $\mathsf{ToCHC}(\mathcal{P}'[\Pi])$ is $\mathcal{T}$-satisfiable. By definition:

$$\begin{aligned}
\mathsf{ToCHC}(\mathcal{P}'[\Pi]) &= wlp(\mathcal{P}'(\mathtt{main}), \top) \wedge \psi_{Procs} \wedge \forall V \cdot (\mathtt{Inv}(x, y) \Leftrightarrow \varphi) \\
wlp(\mathcal{P}'(\mathtt{main}), \top) &= \forall V \cdot wlp(S_1, \mathtt{Inv}(x, y) \wedge \forall V' \cdot (\mathtt{Inv}(x', y') \Rightarrow \\
&\qquad ((e' \Rightarrow wlp(S_2, \mathtt{Inv}(x', y'))) \wedge (\neg e' \Rightarrow wlp(S_3, \top)))))
\end{aligned}$$

As $\forall V \cdot \mathtt{Inv}(x, y) \Leftrightarrow \varphi$, then $\mathsf{ToCHC}(\mathcal{P}'[\Pi])$ is $\mathcal{T}$-satisfiable if and only if $\mathsf{ToCHC}(\mathcal{P}'[\Pi])$ is $\mathcal{T}$-satisfiable after substituting φ for `Inv`. By substituting φ for `Inv` and rewriting $wlp(\mathcal{P}'(\mathtt{main}), \top)$ as a conjunction of CHCs, it follows that $\mathsf{ToCHC}(\mathcal{P}'[\Pi]) \Leftrightarrow \varphi_{Pre} \wedge \varphi_{Tr} \wedge \varphi_{Post} \wedge \varphi_{Procs}$. Therefore, φ is a solution to $(\mathcal{P}, \mathcal{T})$. The other direction is symmetric. ◀

D Proof of Theorem 7

Proof. Let $k = |GV \cup LV|$. The call to `Init` on line 21 of Algorithm 2 has complexity $O(|Locs| + |PE|)$ since `Init` iterates over $\mathsf{Partial}(\mathcal{P})$ and calls `InitBoolReach`, `InitBoolReach` iterates over $Locs$ and PE, and $|Locs| \geq \mathsf{Partial}(\mathcal{P})$. The loop on line 22 of Algorithm 2 requires $O\left(|Locs| \cdot 2^{2k}\right)$ iterations, since each state is visited at most once, there are at most 2^{2k} variable assignments, and exactly $|Locs|$ control-flow locations for each assignment. During each iteration, six procedures are called with the following worst-case runtime complexities:

- Each call to `DoIntraproc` performs $O(|NE|)$ operations since `DoIntraproc` iterates over NE and does $O(1)$ operations per edge in NE.
- Each call to `DoProcs` performs $O(|CE|)$ operations since `DoProcs` iterates over CE and does $O(1)$ operations per edge in CE.

```
1 void main(int x1, int x2) {
2   // Code before loop.
3   S1;
4   // Loop condition and body.
5   while (e) { S2; }
6   // Code after loop.
7   S3;
8 }
```

(a) The input program.

```
1  bool PRED_TEMPLATE Inv(int x1, int x2) {
2    return synth(x1, x2); }
3  void main(int x1, int x2) {
4    S1;
5    assert(Inv(x1, x2));
6    x1 = *; x2 = *;
7    assume(Inv(x1, x2));
8    if (e) {
9      S2;
10     assert(Inv(x1, x2));
11     assume(false);
12   }
13   S3;
14 }
```

(b) The IPS-MP reduction.

Figure 15 A reduction from loop invariant inference to IPS-MP. In both programs, (S_1, S_2, S_3) are statements, and e is a Boolean expression in the language grammar.

- Each call to `DoAssumes` performs $O(|FE|)$ operations since `DoAssumes` iterates over FE and does $O(1)$ operations per edge in FE.
- Each call to `DoAsserts` performs $O(|AE|)$ operations since `DoAsserts` iterates over AE and does $O(1)$ operations per edge in AE.
- Each call to `DoProcSum` iterates over PE. However, `DoProcSum` only processes $(l_{\text{in}}, l_{\text{out}}) \in PE$ if $l_{\text{out}} = l_{\text{wk}}$. Since each function has a single exit location, there is at most one edge in PE such that $l_{\text{out}} = l_{\text{wk}}$. Therefore, each call to `DoProcSum` performs $O(|CE|)$ operations, as once $(l_{\text{in}}, l_{\text{out}})$ is found, `DoProcSum` does $O(1)$ operations, iterates over CE, and does $O(1)$ operations per edge in CE.
- Each call to `DoFuncSum` performs $O(|FE \cup AE|)$ operations since it iterates over both FE and AE, and does $O(1)$ operations per edge in $FE \cup AE$.

Therefore, `BoolSynth` terminates within $O\left(2^{2k} \cdot |Locs| \cdot |NE \cup CE \cup FE \cup AE|\right)$ operations. ◀

E Proof of Theorem 8

Proof. From the correctness of `ComputeBoolReach` (Algorithm 1):

1. `DoInterproc` encodes rule 3 of Def. 2;
2. `DoProcs` encodes rules 4 and 5 of Def. 2;
3. `DoProcSum` encodes rule 6 and maintains rule 5 of Def. 2.

Algorithm 2 introduces three new procedures, that correspond to the rules of Def. 6.

1. `DoAssumes` encodes rule 5 of Def. 6;
2. `DoAsserts` encodes rule 3, and queues work for `DoFunSum`;
3. `DoFunSum` maintains rules 4 and 5 of Def. 6 for items queued in `DoAsserts`.

Following the proof of Algorithm 1, the loop on line 22 computes a fixed point of the equations in Def. 6. This is the least solution that is weaker than the initial assignment to (θ, σ, Π) on line 21. Both θ and σ are initialized according to `ComputeBoolReach`, and follow rule 2 of Def. 2. The initial assignment to Π is Π_0. Therefore, when `Analyze` terminates, (θ, σ, Π) is a least partial program summary such that $\forall p \in \mathsf{Partial}(\mathcal{P}) \cdot \Pi(p) \Rightarrow \Pi'(p)$. ◀

F Proof of Corollary 9

The following proposition was stated informally in Sec. 3.3.

▶ **Proposition 21** ([6]). *If (θ, σ) is the least summary of a Boolean program $\mathcal{P}$, then $\mathcal{P}$ is correct if and only if $\sigma(l_\perp) = \perp$.*

The proof of Cor. 9 follows.

Proof. Assume that $(\mathcal{P}, \varnothing, \Pi_0)$ is an IPS-MP problem for a Boolean program $\mathcal{P}$. By Thm. 8, line 31 computes a least partial program summary (θ, σ, Π) for $\mathcal{P}$ such that $\forall p \in \mathsf{Partial}(\mathcal{P}) \cdot \Pi_0(p) \Rightarrow \Pi(p)$. Furthermore, by Thm. 7, the call on line 31 always terminates. Then, Π solves the IPS-MP instance if and only if $\mathcal{P}$ is safe. By Prop. 21, $\mathcal{P}[\Pi]$ is safe if and only if $\theta(l_\perp) = \perp$. On line 32, if $\theta(l_\perp) = \perp$, then a solution is returned, else a witness to unrealizability is returned. Therefore, `BoolSynth` decides the IPS-MP problem for Boolean programs. ◀

G Proof of Theorem 10

The following proposition was stated informally in Sec. 3.3.

▶ **Proposition 22** ([14]). *A program $\mathcal{P}$ is correct relative to theory $\mathcal{T}$ if and only if $\mathsf{ToCHC}(\mathcal{P})$ has a $\mathcal{T}$-model.*

The proof of Thm. 10 follows.

Proof. Let $\mathsf{CHCBnd}(\mathcal{P}) := \bigwedge_{p \in \mathsf{Partial}(\mathcal{P})} \forall \vec{x} \cdot (\Pi_0(p) \Rightarrow p(\vec{x}))$. Assume that σ is an $\mathcal{F}$-solution to $\mathsf{CHCSynth}(\mathcal{P})$. Since $\mathsf{CHCBnd}(\mathcal{P})$ is a term of $\mathsf{CHCSynth}(\mathcal{P})$, then $\forall p \in \mathsf{Partial}(\mathcal{P}) \cdot \models_{\mathcal{T}} \Pi_0(p) \Rightarrow \Pi(p)$. It then follows by induction on the size of $\mathsf{Partial}(\mathcal{P})$ that $\mathcal{P}[\Pi]$ is correct.

- **Hypothesis**. For some $k \geq 0$, if $|\mathsf{Partial}(\mathcal{P})| \leq k$ and $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ has an $\mathcal{F}$-solution, then $\mathcal{P}$ is correct.
- **Base Case**. If $|\mathsf{Partial}(\mathcal{P})| = 0$, then $\mathcal{P}$ is correct by Prop. 22.
- **Inductive Case**. Assume that $|\mathsf{Partial}(\mathcal{P})| = k + 1$, $\mathsf{CHCSynth}(\mathcal{P})$ has an $\mathcal{F}$-solution σ, and the inductive hypothesis holds up to k. Let $p \in \mathsf{Partial}(\mathcal{P})$ and $\mathcal{P}' = \mathcal{P}[p \leftarrow \sigma(p)]$. By the definition of an interpretation, σ is also a solution to $\mathsf{ToCHC}(\mathcal{P}) \wedge \mathsf{CHCBnd}(\mathcal{P}) \wedge \forall \vec{x} \cdot p(\vec{x}) \Leftrightarrow \sigma(p)$. Furthermore, $\forall \vec{x} \cdot \Pi_0(p) \Rightarrow p(\vec{x})$ is subsumed by $\forall \vec{x} \cdot p(\vec{x}) \Leftrightarrow \sigma(p)$. Therefore, σ is an $\mathcal{F}$-solution to $\mathsf{CHCSynth}(\mathcal{P}')$. By hypothesis, $\mathcal{P}'[\Pi]$ is correct. Since $\Pi(p) = \sigma(p)$, then $\mathcal{P}'[\Pi] = \mathcal{P}[p \leftarrow \sigma(p)][\Pi] = \mathcal{P}[\Pi]$. Therefore, $\mathcal{P}[\Pi]$ is also correct.

Therefore, Π is a solution to $(\mathcal{P}, \mathcal{T}, \Pi_0)$. ◀

H Proof of Theorem 11

The following proposition was stated informally in Sec. 3.3.

▶ **Proposition 23** ([14]). *$\mathsf{ToCHC}(\mathcal{P})$ is a CHC conjunction.*

The proof of Thm. 11 follows.

Proof. Assume for the intent of contradiction that Π is a solution to $(\mathcal{P}, \mathcal{T}, \Pi_0)$. Then $\mathcal{P}[\Pi]$ is correct relative to $\mathcal{T}$, since Π is a solution. Then by Prop. 23, $\mathsf{ToCHC}(\mathcal{P}[\Pi])$ is $\mathcal{T}$-satisfiable, since $\mathcal{P}[\Pi]$ is correct relative to $\mathcal{T}$. By definition:

$$\mathsf{ToCHC}(\mathcal{P}[\Pi]) = \mathsf{ToCHC}(\mathcal{P}) \wedge \left(\bigwedge_{p \in \mathsf{Partial}(\mathcal{P})} \forall \vec{x} \, (\Pi(p) \Leftrightarrow p(\vec{x})) \right)$$

Then, $\mathsf{ToCHC}(\mathcal{P}) \wedge \left(\bigwedge_{p \in \mathsf{Partial}(\mathcal{P})} \forall \vec{x} \, (\Pi_0(p) \Rightarrow p(\vec{x})) \right)$ has a $\mathcal{T}$-solution, since Π is a solution to $(\mathcal{P}, \mathcal{T}, \Pi_0)$ and therefore satisfies $\forall p \in \mathsf{Partial}(\mathcal{P}) \cdot \models_{\mathcal{T}} \Pi(p) \Rightarrow \Pi_0(p)$. Then $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ is $\mathcal{T}$-satisfiable. By contradiction, $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is unrealizable. ◀

I Proof of Theorem 12

Proof. By Prop. 23, $\mathsf{ToCHC}(\mathcal{P})$ is a CHC conjunction. For each $p \in \mathsf{Partial}(\mathcal{P})$, $\Pi_0(p) \Rightarrow p(\vec{x})$ is a CHC, since $\Pi_0(p)$ is quantifier-free and p is a predicate symbol. Therefore, $\mathsf{CHCSynth}(\mathcal{P}, \Pi_0)$ is a CHC conjunction. ◀

J Proof of Theorem 14

The following proposition was stated informally in Sec. 5.2.

▶ **Proposition 24** ([49]). *The halting problem is undecidable for 2-counters.*

The proof of Thm. 14 follows.

Proof. Assume for the intent of contradiction that IPS-MP is decidable for linear integer arithmetic. Let $\mathcal{V} = \{x, y, z\}$, Σ be the signature of linear integer arithmetic, and $\mathcal{T}$ be the theory of linear integer arithmetic. Every 2-counter machine can be encoded in $\mathsf{Progs}(\Sigma, \mathcal{V})$ as follows:

1. The two integer counters are x and y.
2. The program counter is z.
3. The body of $\mathcal{P}(\texttt{main})$ is a `while` loop with loop condition `true`.
4. The body of the `while` loop is a sequence of `if`-`else` statements that maps each value of z to an instruction.
5. The instruction `inc(x)` is: `x = x + 1; z = z + 1;`
6. The instruction `dec(x)` is: `x = x - 1; z = z + 1;`
7. The instruction `jump(x, i)` is: `if (x==0) { z=i; } else { z=z+1; }`
8. The instruction `halt()` is: `assert(false);`

If $\mathcal{P} \in \mathsf{Progs}(\Sigma, \mathcal{V})$ is a 2-counter machine (following the above encoding), then $\mathcal{P}$ halts if and only if $\mathcal{P}$ violates an assertion. Furthermore, $|\mathsf{Partial}(\mathcal{P})| = 0$. Let Π_0 be the trivial function from $\mathsf{Partial}(\mathcal{P})$ to $\mathsf{QFFml}(\Sigma, \mathcal{V})$. Then the IPS-MP problem $(\mathcal{P}, \mathcal{T}, \Pi_0)$ has a solution if and only if $\mathcal{P}$ halts. Since IPS-MP is decidable, then the halting problem for 2-counter machines is decidable. However, the halting problem is undecidable for 2-counter machines by Prop. 24. Then by contradiction, IPS-MP is undecidable for the theory of integer linear arithmetic. ◀

K Proof of Theorem 16

Proof. Let Π be a solution $(\mathcal{P}', \Pi_\perp, \mathcal{T})$. This is true if and only if $\mathcal{P}'[\Pi]$ is correct relative to $\mathcal{T}$. By Prop. 22, this is true if and only if $\mathsf{ToCHC}(\mathcal{P}'[\Pi])$ is $\mathcal{T}$-satisfiable. By definition

$$\mathsf{ToCHC}(\mathcal{P}'[\Pi]) = wlp(\mathcal{P}'(\texttt{main}), \top) \wedge \mathsf{ToCHC}(\mathcal{P}) \wedge \forall V \cdot (\texttt{Inv}(x, y) \Leftrightarrow \varphi)$$

$$wlp(\mathcal{P}'(\texttt{main}), \top) = \forall V \cdot (((\texttt{br} = 0) \Rightarrow \tau_0) \wedge \cdots \wedge ((\texttt{br} = 3) \Rightarrow \tau_3))$$

where τ_i is the WLP of the i-th branch of $\mathcal{P}'(\texttt{main})$. Since `br` does not appear in any τ_i, then $wlp(\mathcal{P}'(\texttt{main}), \top)$ is equisatisfiable with $(\forall V \cdot \tau_0) \wedge \cdots \wedge (\forall V \cdot \tau_3)$. Since $\forall V \cdot (\texttt{Inv}(x, y) \Leftrightarrow \varphi)$, $\mathsf{ToCHC}(\mathcal{P}'[\Pi])$ is $\mathcal{T}$-satisfiable if and only if $\mathsf{ToCHC}(\mathcal{P}'[\Pi])$ is $\mathcal{T}$-satisfiable after substituting φ for `Inv`. Observe that, after substituting φ for `Inv` and simplifying to CHC conjunctions:

$$\tau_0 = \psi_{Init} \qquad \tau_1 = \psi_{Closure1} \wedge \forall V \cdot (\varphi \Rightarrow \mathsf{f}_{pre}(x, y, a))$$

$$\tau_2 = \psi_{Closure2} \wedge \forall V \cdot \big(\varphi \Rightarrow \mathsf{g}_{pre}(x, y, a, b)\big) \qquad \tau_3 = \psi_{Sufficient} \wedge \forall V \cdot (\varphi \Rightarrow \mathsf{func}_{pre}(x, y, a))$$

As a direct result, $\mathsf{ToCHC}(\mathcal{P}[\Pi']) \Leftrightarrow \psi_{Init} \wedge \psi_{Closure1} \wedge \psi_{Closure2} \wedge \psi_{Sufficient} \wedge \psi_{Sum}$. Therefore, φ is a solution to $(\mathcal{P}, \mathcal{T})$. The other direction is symmetric. ◀

L Proof of Theorem 18

Proof. Let Π be a solution $(\mathcal{P}', \Pi_0, \mathcal{T})$. This is true if and only if $\mathcal{P}'[\Pi]$ is correct relative to $\mathcal{T}$. By Prop. 22, this is true if and only if $\mathsf{ToCHC}(\mathcal{P}'[\Pi])$ is $\mathcal{T}$-satisfiable. By definition

$$\mathsf{ToCHC}(\mathcal{P}'[\Pi]) = wlp(\mathcal{P}'(\texttt{main}), \top) \wedge \psi_{Process} \wedge \forall V \cdot (\texttt{Inv}(l, s, r) \Leftrightarrow \varphi)$$

$$wlp(\mathcal{P}'(\texttt{main}), \top) = \forall V \cdot (((\texttt{br} = 0) \Rightarrow \tau_0) \wedge ((\texttt{br} \neq 0) \Rightarrow \tau_1)),$$

where τ_i is the WLP of the i-th branch of $\mathcal{P}'(\texttt{main})$. Since $\texttt{br}$ does not appear in τ_0 or τ_1, then $wlp(\mathcal{P}'(\texttt{main}), \top)$ is equisatisfiable with $(\forall V \cdot \tau_0) \wedge (\forall V \cdot \tau_1)$. By definition of wlp:

$$\tau_0 = (\texttt{Inv}(l, s, r) \wedge \texttt{Inv}(r, i, l)) \Rightarrow \\ \big(\texttt{tr}_{pre}(l, s, r) \wedge \big(\texttt{tr}_{sum}(l, s, r, l', s', r') \Rightarrow \big(\texttt{Inv}(l', s', r') \wedge \texttt{Inv}(r', i, l')\big)\big)\big)$$

Since $\forall V \cdot (\texttt{Inv}(x, y)) \Leftrightarrow \varphi$, $\mathsf{ToCHC}(\mathcal{P}'[\Pi])$ is $\mathcal{T}$-satisfiable if and only if $\mathsf{ToCHC}(\mathcal{P}'[\Pi])$ is $\mathcal{T}$-satisfiable after substituting φ for $\texttt{Inv}$. Observe that, after substituting φ for $\texttt{Inv}$ and simplifying to CHC conjunctions:

$$\tau_0 = \psi_{Closure} \wedge \psi_{Inf} \wedge \forall V \cdot (\varphi \wedge \varphi_{Inf} \Rightarrow \texttt{tr}_{pre}(l, s, r)) \qquad \tau_1 = \psi_{Adequate}$$

Then $\mathsf{ToCHC}(\mathcal{P}[\Pi']) \Leftrightarrow \psi_{Closure} \wedge \psi_{Inf} \wedge \psi_{Adequate} \wedge \psi_{Sum}$. Since the template for $\texttt{Inv}$ is $\texttt{init}$, then also $\models_{\mathcal{T}} \texttt{init}(l, s, r) \Rightarrow \varphi$. Therefore, φ is a solution to $(\mathcal{P}, \mathcal{T})$. The other direction is symmetric. ◀