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.
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 SEAHO∗ [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 SEAHO∗ (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 SEAHO∗. The results obtained from SEAHO∗ 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 SEAHO∗ 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 SEAHO∗. The goal of this paper is to extend SEAHO∗ 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 SEAHO∗ 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 SEAHO∗’s invariant inference capabilities. However, SEAHO∗ 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 SEAHO∗, then SEAHO∗ could instruct the underlying verification algorithm to infer all invariants automatically. Inspired by this hypothesis, we first implemented compositional invariant synthesis in SEAHO∗, 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., SEAHO∗) 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.
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 verifier3, 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.
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
Figure 3 A simple example of the IPS-MP problem.

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 \( \text{synth} \) 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:

- \( \text{post}_{\text{weak}}(x, y) = (x = y) \)
- \( \text{post}_{\text{strong}}(x, y) = (y > 0 \land x = y) \)

Intuitively, each call to \( \text{Post} \) under \( \text{assume} \) provides an implicit constraint on the weakest possible synthesized solution. Likewise, each call to \( \text{Post} \) under \( \text{assert} \) 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, \( \text{Post} \) cannot be mapped to \( \text{false} \), since this violates the assertion on line 8. If \( \text{Post} \) is not \( \text{false} \), 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 \( \text{Post} \) 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 \( \text{synth}_{\text{witness}}(x, y) = (x = 0 \land y = 1) \).

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, \( \text{pos} \), and the maximum value, \( \text{max} \). The initialization procedure on lines 3–6 first ensures that \( \text{max} \) is positive, and then sets the current value to 0 and the maximum value to \( \text{max} \). 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 \text{true} is returned, else the current value is unchanged and \text{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 \text{true} on line 17, and capacity always returns a positive value on line 19. Verifying these claims
**Inductive Predicate Synthesis Modulo Programs (Extended)**

```java
class Counter {
    int max; int pos;
    Counter(int _max) {
        assume(_max > 0);
        max = _max;
        pos = 0;
    }
    void reset() { pos = 0; }
    int capacity() {
        return max - pos;
    }
    bool increment() {
        if (pos >= max) return false;
        pos += 1;
        return true;
    }
}
```

```java
bool PRED_TEMPLATE CInv(int m, int p) {
    return synth(m, p);
}
void main(void) {
    if (*) {
        Counter o(*);
        assert(CInv(o.max, o.pos));
    } else if (*) {
        Counter o = *;
        assume(CInv(o.max, o.pos));
        o.reset();
        assert(CInv(o.max, o.pos));
    } else if (*) {
        Counter o = *;
        assume(CInv(o.max, o.pos));
        o.increment();
        assert(CInv(o.max, o.pos));
    } else {
        Counter o = *;
        assume(CInv(o.max, o.pos));
        drain(o);
    }
}
```

(a) The original program.

(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 \((0 < o.\text{max}) \land (0 \leq o.\text{pos} \leq o.\text{max})\), 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 \((0 \leq \text{max} - \text{pos})\).

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 \(0 \leq \text{max} - \text{pos}\)). Therefore, `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 \((m > 0) \land (p \leq m)\) for the hole in `CInv`. To prove the correctness of `drain`, a synthesizer may also infer the invariant \((0 \leq o.\text{pos} \land o.\text{pos} \leq o.\text{max})\) for the loop on line 16 of Figure 4a.
Consider the array-manipulating program in Figure 5a. This program initializes the array \( \text{data} \), and then performs an unbounded sequence of updates to the cells of \( \text{data} \) while maintaining the maximum element of \( \text{data} \) in \( \text{max} \). A special index, stored by \( \text{sid} \), remains unchanged during execution. On lines 2–4, \( \text{data} \) is allocated and then zero-initialized. On line 5, \( \text{max} \) is set to 0, since the maximum element of a zero-initialized array is 0. On line 6, \( \text{sid} \) is set to an arbitrary index in \( \text{data} \). 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 \( \text{sid} \), then the corresponding cell in \( \text{data} \) is incremented by 1 (line 11). If the cell is incremented, then \( \text{max} \) is updated accordingly (line 12). Note that Figure 5a can be thought of as a simplified smart contract, where \( \text{data} \) is an address mapping, \( \text{sid} \) 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 \( \text{data} \). The first property is that every cell of \( \text{data} \) is at most \( \text{max} \). The second property is that \( \text{data}[\text{sid}] \) 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 \( \text{data} \) is initially zero, and after increasing the value of a cell, \( \text{max} \) is updated accordingly. However, verifying these properties is challenging, since \( \text{data} \) has an arbitrary number of cells. One solution to this problem is to compute a summary for each cell of \( \text{data} \), with respect to \( \text{max} \) and \( \text{sid} \), and independent of \( \text{data} \)’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

\begin{lstlisting}[language=C++]
void main(int sz) {
    int * data = new int [sz];
membset(data, 0, sz * sizeof(int));
    int max = 0; int sid = *
    assume(0 <= sid && sid < sz);
    while (true) {
        int id = *
        assume(0 <= id && id < sz);
        int v = data[id];
        if (id != sid) {v += 1;}
        if (v > max ) { max = v; }
        data[id] = v; }
}

bool PRED_TEMPLATE Inv3 (int m, int v) {
    if (m == 0 && v == 0) { return true ; }
    else { return synth (m, v); } }
bool PRED_TEMPLATE Inv4 (int m, int v) {
    if (m == 0 && v == 0) { return true ; }
    else { return synth (m, v); }}

void main (int sid) {
    int max = 0;
    while (true) {
        int id = *
        int x = *
        assume(id != x);
        int v = *; int u = *
        if (id == sid ) { assume ( Inv3 (max ,v)); }
        else { assume ( Inv4 (max ,v)); }
        if (x == sid ) { assume ( Inv3 (max ,u)); }
        else { assume ( Inv4 (max ,u)); }
        assert (v <= max );
        if (id == sid ) { assert (v == 0); }
        // Update.
        if (id != sid ) { v += 1; }
        if (v > max ) { max = v; }
        // data [id] = v; }
    }
}
\end{lstlisting}
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 \( \text{data}[i] \) is updated, then \( \text{data}[j] \) 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 \( \text{id} \) is selected, as in the original program. However, on lines 11–12, a second, distinct index named \( \text{x} \) 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 \( \text{data}[\text{id}] \) and \( \text{data}[\text{x}] \). 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 \( \text{data}[\text{id}] \). Note that lines 2–4 of Figure 5a do not appear in Figure 5b since the compositional invariant abstracts away the contents of \( \text{data} \). 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 \( \text{data} \) depends on whether the index is \( \text{sid} \). This suggests that the compositional invariant has two cases that branch on whether \( \text{id} \) equals \( \text{sid} \), namely \(((\text{id} = \text{sid}) \land \text{inv3}(\text{max}, v)) \lor ((\text{id} \neq \text{sid}) \land \text{inv4}(\text{max}, v))\). In the IPS-MP encoding, both \( \text{inv3} \) and \( \text{inv4} \) 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 \( \text{inv3}(\text{max}, 0) \) and \( \text{inv4}(\text{max}, 0) \) 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 \( \text{inv3} \), and \((0 <= v) \land (v <= \text{max})\) for the hole in \( \text{inv4} \). By substitution, \(((\text{id} = \text{sid}) \land (v = 0)) \lor ((\text{id} \neq \text{sid}) \land (0 <= v) \land (v <= \text{max}))\). To verify \( \text{main} \), a synthesizer may also infer the invariant \((0 <= \text{max})\) 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 \( \text{view} \) on line 3, and the transition relation of each process is given by \( \text{tr}\) 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

---

For simplicity, \( \text{tr} \) 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;
}

bool PRED_TEMPLATE RInv(
    Lock l, State s, Lock r) {
    if (l == Free && r == l && s == Try)
        return true;
    return synth(l, s, r);
}

void main(struct View v) {
    State otr = *
    assume (RInv(v.left, v.st, v.right));
    assume (RInv(v.right, otr, v.left));
    assert (RInv(v.left, v.st, v.right));
    assert (RInv(v.right, otr, v.left));
    else {
        assume (RInv(v.left, v.st, v.right));
        bool held = v.left == Left && v.right == Right;
        assert (v.st != Critical || held);
    }
}

(a) The process.
(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 \((v.lhs \neq Left) \lor (v.rhs \neq Right) \Rightarrow (v.st \neq Critical)\), 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, 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, \((s \neq Try) \Rightarrow (l = Left \land r = Right)\).

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
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 \texttt{false}, are permitted. Since IPS-MP only places functional requirements on its solutions, this case is trivial and returning \texttt{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 \texttt{true} subsumes all possible assertions. Therefore, this case is also trivial and returning \texttt{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 \( \Sigma \) 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
Definition 1. Let $\Sigma_{\text{Bool}}$ denote a Boolean signature. A Boolean program is a tuple $(\text{Locs}, GV, LV, E)$ 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 $\text{main} \in \text{Locs}$;
2. GV and LV are disjoint sets of local and global variables (respectively);
3. $NE \subseteq \text{Locs} \times \text{QFFml}(\Sigma_{\text{Bool}}, V \cup V') \times \text{Locs}$ is a set of normal edges, $CE \subseteq \text{Locs} \times \text{Locs}$ is a set of call edges, $FE \subseteq \text{Locs} \times \text{Locs} \times \text{Locs}$ is a set of (partial predicate) call-under-assert edges, $AE \subseteq \text{Locs} \times \text{Locs} \times \text{Locs}$ is a set of (partial predicate) call-under-assume edges, and $PE \subseteq \text{Locs} \times \text{Locs}$ 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 $(l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in CE$, then $l_{\text{ret}}$ is reachable from $l_{\text{call}}$ by executing the procedure with entry location $l_{\text{in}}$;
5. If $(l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in FE$, then $l_{\text{ret}}$ is reachable from $l_{\text{in}}$ by assuming the partial predicate with entry location $l_{\text{call}}$ and if $(l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in AE$, then $l_{\text{ret}}$ is reachable from $l_{\text{in}}$ by asserting the partial predicate with entry location $l_{\text{call}}$.

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

Constrained Horn Clauses (CHCs) are a fragment of FOL determined by a FO-signature $\Sigma$ and an set of predicates $P$. A CHC is a sentence of the form $\forall V \cdot \varphi \land p_1(x_1) \land \cdots \land p_k(x_k) \Rightarrow h(y)$, where $\varphi \in \text{QFFml}(\Sigma, V)$ and $\{p_1, \ldots, p_k, h\} \subseteq P$. For program semantics, it is useful to use $\varphi'$ to denote the value after a program transition and $\text{keep}(W) := \bigwedge_{w \in W} w = v'$ to denote that values $W \subseteq V$ are unchanged during a transition. Given a set of variables $V = \{v_1, \ldots, v_n\} \subseteq V$, the set of variables $\{v'_1, \ldots, v'_n\}$ is denoted $V'$. Likewise, given a formula $\varphi$ over the variables in $V$, the formula $\varphi[v_1/v'_1] \cdots [v_n/v'_n]$ over $V'$ is denoted $\varphi'$.

### 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 $\Sigma$, with variables from a set $V$. That is, each expression is of the form $\text{Term}(\Sigma, V)$. The set of all programs in the language is denoted $\text{Progs}(\Sigma, V)$. For simplicity, types are omitted. In this language, a program has one or more procedures, with execution starting from $\text{main}$. Each procedure is written in an imperative language, including loops and procedure calls. The language is extended with a non-deterministic assignment (i.e., $\star$), and verification statements $\text{assume}$ and $\text{assert}$. The expressions in assume and assert can be either from $\text{QFFml}(\Sigma, V)$ or a call to a pure Boolean procedure, called a predicate. Predicates may only be called within assume or assert statements. Given a program $P \in \text{Progs}(\Sigma, V)$, $\text{Procs}(P)$ denotes the procedures in $P$. A special case is when all variables are Boolean.

Definition 1. Let $\Sigma_{\text{Bool}}$ denote a Boolean signature. A Boolean program is a tuple $(\text{Locs}, GV, LV, E)$ 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 $\text{main} \in \text{Locs}$;
2. GV and LV are disjoint sets of local and global variables (respectively);
3. NE $\subseteq$ Locs $\times$ QFFml($\Sigma_{\text{Bool}}$, V $\cup$ V') $\times$ Locs is a set of normal edges, CE $\subseteq$ Locs $\times$ Locs is a set of call edges, FE $\subseteq$ Locs $\times$ Locs $\times$ Locs is a set of (partial predicate) call-under-assert edges, AE $\subseteq$ Locs $\times$ Locs $\times$ Locs is a set of (partial predicate) call-under-assume 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 $(l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in CE$, then $l_{\text{ret}}$ is reachable from $l_{\text{call}}$ by executing the procedure with entry location $l_{\text{in}}$;
5. If $(l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in FE$, then $l_{\text{ret}}$ is reachable from $l_{\text{in}}$ by assuming the partial predicate with entry location $l_{\text{call}}$ and if $(l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in AE$, then $l_{\text{ret}}$ is reachable from $l_{\text{in}}$ by asserting the partial predicate with entry location $l_{\text{call}}$;
The Weakest Liberal Precondition (WLP) transformer gives logical semantics to imperative programs \([23]\). We write \(\text{wlp}(S, Q)\) for the WLP of a statement \(S\) with respect to a post-condition \(Q\). The WLP transformer for \(\text{Progs}(\Sigma, \mathcal{V})\) is standard and can be found in Appx. A. Note that in this transformation, the \(\text{loop}_{\text{ln}}\) predicate is an invariant for a loop at line \(\text{ln}\).

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

Efficient procedures exist to prove that Boolean programs are correct. For example, \textit{program summarization} simultaneously computes a summary \(\theta\) from control-flow locations to input-to-reachable-state relations, and a summary \(\sigma\) from procedures to input-output relations. For a location \(l \in \text{Locs}\), if \(\theta(l) = \bot\), then \(l\) is unreachable. Therefore, a Boolean program \(\mathcal{P}\) is correct if and only if \(\theta(l_{\bot}) = \bot\) in the least summary of \(\mathcal{P}\) [6].
summarization is defined in Def. 2. The algorithm to compute \( \theta \) is presented in full, for reuse in Sec. 5.1. For presentation, \( \text{elim}(\varphi, W) \) denotes the existential elimination of \( W \) in \( \varphi \).

**Definition 2** ([6]). A Boolean program summary for \((L, GV, LV, E)\), where \(E = (NE, CE, \emptyset, \emptyset, PE)\) is a tuple \((\theta, \sigma)\) such that \(Q = \text{QFFml}(\Sigma_{\text{Bool}}, V \cup V')\), \(V = GV \cup LV\) and the following hold:

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

**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 finitestate 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 \(P\) 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 \(P[p \leftarrow e]\). The set of all partial

---

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

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 unrealizability.

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.

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

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

Assume that $(\mathcal{P}, T, \Pi_0)$ is an IPS-MP problem with a solution $\Pi$. With respect to the IPS-MP overview in Figure 2, $\mathcal{P}$ is a program with specifications, $\Pi_0$ is a collection of predicate templates, and $\Pi$ 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 $(\mathcal{P}, \Pi_0, T)$. The program $\mathcal{P}$ is given by lines 7–29 of Figure 5b. Then $\text{Partial}(\mathcal{P}) = \{\text{Inv3}, \text{Inv4}\}$, since $\text{Inv3}$ and $\text{Inv4}$ are called on lines 15–16, but lack full implementations. From lines 1–6, $\Pi_0(\text{Inv3}) = \Pi_0(\text{Inv4}) = (m = 0 \land v = 0)$. Now, recall from Sec. 2.3 that all variables in Figure 5b are integer linear arithmetic. Therefore, $T$ is the theory of integer linear arithmetic. A solution to $(\mathcal{P}, \Pi_0, T)$ is $\Pi$ such that $\Pi(\text{Inv3}) = (v = 0)$ and $\Pi(\text{Inv4}) = (0 \leq v \land v \leq m)$.

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

[Code example]

```
bool Post(int x, int y) {
    return x==y;
}
void main(int y) {
    assume(y>0); int x=0;
    for (int i=0; i<y; ++i) { x+=1; }
    assert(Post(x, y));
    x++; y++; assume(Post(x, y));
}
```

```
bool Post(int x, int y) {
    return (y>0) && (x==y);
}
void main(int y) {
    assume(y>0); int x=0;
    for (int i=0; i<y; ++i) { x+=1; }
    assert(Post(x, y));
    x++; y++; assume(Post(x, y));
}
```
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 $P = (\text{Locs}, GV, LV, (NE, CE, FE, AE, PE))$ be a Boolean program. A partial program summary for $P$ is a tuple $(\theta, \sigma, \Pi)$ such that:

1. $\Pi : \textit{Partial}(P) \rightarrow QF\text{Fml}(\Sigma_{\text{Bool}}, GV);$
2. $(\theta, \sigma)$ is a program summary for $(\text{Locs}, GV, LV, (NE, CE, \varnothing, \varnothing, PE));$
3. $\forall (l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in AE : \theta(l_{\text{call}}) \Rightarrow \Pi'(l_{\text{in}});$
4. $\forall (l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in AE : \theta(l_{\text{call}}) \Rightarrow \theta(l_{\text{ret}});$
5. $\forall (l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in FE : \theta(l_{\text{call}}) \wedge \Pi'(l_{\text{in}}) \Rightarrow \theta(l_{\text{ret}}).$

The rules of Def. 6 follow directly from the preceding discussion. Rule 2 ensures that $(\theta, \sigma)$ is a program summary for $P$ after discarding all calls to partial predicates. Rules 3 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 $P = (\text{Locs}, GV, LV, E)$ with $E = (NE, CE, FE, AE, PE)$ be a Boolean program. Then for each input $(P, \Pi_0)$, Analyze of Algorithm 2 terminates in $O(n^2m \cdot |\text{Locs}|)$ symbolic Boolean operations where $n = 2^{\varnothing \cup LV}$ is the number of variable assignments and $m = |\text{Locs}|$ 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.

```
1 var (θ, σ, Π): // A partial program summary
2 Func DoAssumes(V, LV, PE, FE, l_wk, s WK): for (l wk, l m, l wk) ∈ FE do
3     s in ← {elim(s wk, V ∪ LV'[V' / V])};
4     UpdateReach(l m, Π(l m) ∧ s in);
5 for (l wk, l m, l wk) ∈ AE do
6     UpdateReach(l m, Π(l m) ∧ s wk);
7     UpdateReach(l wk, l wk ∧ s wk);
8     l wk ← l wk ∧ s wk;
9     if l wk ∈ Partial(Π), then
10        Π(l wk) ← Π(l wk) ∨ s wk;
11        for (l calt, l wk, l wk) ∈ FE ∪ AE do
12           UpdateReach(l calt, θ(l calt) ∧ s wk);
13        end
14 end
15 Func DoAssumes(V, LV, PE, FE, l wk, s wk): for (l wk, l wk, l wk) ∈ FE do
16     s in ← {elim(s wk, V ∪ LV'[V' / V])};
17     UpdateReach(l wk, Π(l wk) ∧ s in);
18     for (l wk, l wk, l wk) ∈ AE do
19     UpdateReach(l wk, Π(l wk) ∧ s wk);
20     UpdateReach(l wk, l wk ∧ s wk);
21     if l wk ∈ Partial(Π), then
22        Π(l wk) ← Π(l wk) ∨ s wk;
23        for (l calt, l wk, l wk) ∈ FE ∪ AE do
24           UpdateReach(l calt, θ(l calt) ∧ s wk);
25        end
26 end
27 Func DoAssumes(V, LV, PE, FE, l wk, s wk): for (l wk, l wk, l wk) ∈ FE do
28     s in ← {elim(s wk, V ∪ LV'[V' / V])};
29     UpdateReach(l wk, Π(l wk) ∧ s in);
30     for (l wk, l wk, l wk) ∈ AE do
31     UpdateReach(l wk, Π(l wk) ∧ s wk);
32     if l wk ∈ Partial(Π), then
33        Π(l wk) ← Π(l wk) ∨ s wk;
34        for (l calt, l wk, l wk) ∈ FE ∪ AE do
35           UpdateReach(l calt, θ(l calt) ∧ s wk);
36        end
37 end
38 Func Init(Locs, PE, Π): Init(InitReach(Locs, PE));
39     for l ∈ Partial(Π) do Π(l) ← Π0(l);
41     V ← GV ∪ LV; Init(Locs, P, Π);
42     while ∃ l wk ∈ Locs, W(l wk) # ⊥ do
43        s wk ← W(l wk); W(l wk) ← ⊥;
44        DoIntraproc(V, N, l wk, s wk);
45        DoProcSum(V, LV, P, C, l wk, s wk);
46        DoAssumes(V, LV, P, F, l wk, s wk);
47        DoProcSum(V, LV, P, A, l wk, s wk);
48        end
49 Func BoolSynth(P, Π): Analyze(P, Π);
50     if θ0(p) = ⊥ then return (✓, Π);
51     else return (✗, Π);
```

Theorem 8. Let P = (Locs, GV, LV, E) be a Boolean program and Π be a collection of predicate templates for P. Analyze of Algorithm 2 computes a least partial program summary, (θ, σ, Π), for P such that ∀ p ∈ Partial(Π) · Π0(p) ⇒ Π(p).


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 (P, T, Π) is an IPS-MP problem. Recall that P ∈ Progs(F, V) where F is the FO-fragment of pure program expressions. A logical encoding of (P, T, Π) is given by:

\[
\text{CHCSynth}(P, Π) := \text{ToCHC}(P) \land \left( \bigwedge_{P \in \text{Partial}(Π)} \forall \vec{x} \cdot (Π(p) \Rightarrow p(\vec{x})) \right)
\]

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

Theorem 10. Let Σ be a first-order signature, V be a set of variable symbols, F = QFFmol(Σ, V), P ∈ Progs(Σ, V), and (P, T, Π) be an IPS-MP problem. If σ is an F-solution
to \( \text{CHCSynth}(\mathcal{P}, \Pi_0) \) relative to \( \mathcal{T} \), then \( \Pi : \text{Partial}(\mathcal{P}) \rightarrow \mathcal{F} \) such that \( \Pi : p \mapsto \sigma(p) \) is a solution to \( (\mathcal{P}, \mathcal{T}, \Pi_0) \).

\[ \text{Theorem 11.} \] If \( (\mathcal{P}, \mathcal{T}, \Pi_0) \) is an IPS-MP problem and \( \text{CHCSynth}(\mathcal{P}, \Pi_0) \) is \( \mathcal{T} \)-unsatisfiable, then \( (\mathcal{P}, \mathcal{T}, \Pi_0) \) is unrealizable.

\( \text{CHCSynth}(\mathcal{P}, \Pi_0) \) strengthens \( \text{ToCHC}(\mathcal{P}) \) by adding additional CHCs. Since \( \text{ToCHC}(\mathcal{P}) \) is a conjunction of CHCs, then \( \text{CHCSynth}(\mathcal{P}, \Pi_0) \) is also a conjunction of CHCs. Therefore, a CHC solver can check the satisfiability and unsatisfiability of \( \text{CHCSynth}(\mathcal{P}, \Pi_0) \). 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).

\[ \text{Theorem 12.} \] \( \text{CHCSynth}(\mathcal{P}, \Pi_0) \) is a CHC conjunction.

\[ \text{Example 13.} \] 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 \( \Pi_0 : \text{CInv} \rightarrow \bot \). In this example we let \( \mathcal{F} \) be the fragment of linear inequalities of the variables \( \{m, p\} \), where \( m \) and \( p \) are the arguments to \( \text{CInv} \). Then our goal is to find an expression \( e \in \mathcal{F} \) such that \( \mathcal{P}[\text{CInv} \leftarrow e] \) 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 \( \text{CHCSynth}(\mathcal{P}, \Pi_0) \). To construct \( \text{CHCSynth}(\mathcal{P}, \Pi_0) \) we must first construct the term \( \text{ToCHC}(\mathcal{P}) \). Recall that \( \text{ToCHC}(\mathcal{P}) \) encodes verification conditions for the program \( \mathcal{P} \). Since \( \mathcal{P} \) is open (\( \text{CInv} \) is unimplemented), then \( \text{CInv} \) will be an unknown in \( \text{ToCHC}(\mathcal{P}) \). According to Sec. 3.3, \( \text{ToCHC}(\mathcal{P}) \) will consist of the verification conditions for \( \mathcal{P}[\text{main}] \), along with a summary for each function in \( \mathcal{P} \). We begin by constructing a summary for each method from the \text{Counter} object in \( \mathcal{P} \). As described in Sec. 3.3, each predicate \( f_{\text{pre}}(x) \) collects the inputs \( x \) to a function \( f \), and each predicate \( f_{\text{sum}}(x, e) \) each argument \( x \) to a return value \( e \). For simplicity, we encode object state by passing member fields as arguments and return values. Redundant declarations are omitted.

\[
\varphi_{\text{Counter}} := \forall m \cdot \text{Counter}_{\text{pre}}(m) \Rightarrow ((m > 0) \Rightarrow \text{Counter}_{\text{sum}}(m, m, 0))
\]

\[
\varphi_{\text{Reset}} := \forall m \cdot \forall p \cdot \text{reset}_{\text{pre}}(m, p) \Rightarrow \text{reset}_{\text{sum}}(m, p, m, 0)
\]

\[
\varphi_{\text{Cup}} := \forall m \cdot \forall p \cdot \text{capacity}_{\text{pre}}(m, p) \Rightarrow \text{capacity}_{\text{sum}}(m, p, m - p, 0)
\]

\[
\varphi_{\text{Inc}} := \forall m \cdot \forall p \cdot \text{increment}_{\text{pre}}(m, p) \Rightarrow ((p \geq m) \Rightarrow \text{increment}_{\text{sum}}(m, p, 1, T))
\]

Next, we construct a summary for the function \( \text{drain} \). Note that, unlike the methods of \text{Counter}, the function \( \text{drain} \) 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 \( \text{loop}_{13} \). In our example, the loop at Line 16 of Figure 4a is associated with a loop invariant \( \text{loop}_{13} \). Then the summary of \( \text{drain} \) is as follows.

\[
\varphi_{\text{Exit}} := \text{capacity}_{\text{pre}}(p', m') \cdot \forall z \cdot (\text{capacity}_{\text{sum}}(p', m', z) \Rightarrow (z \land \text{drain}_{\text{sum}}(p, p', m'))) \]

\[
\varphi_{\text{Loop}} := \text{loop}_{13}(p, m, z) \land
\]

\[
((\text{loop}_{13}(p, m, z) \land z > 0) \Rightarrow (\text{capacity}_{\text{pre}}(p, m) \land \forall z \cdot (\text{capacity}_{\text{sum}}(p, mz) \Rightarrow \text{loop}_{13}))) \land
\]

\[
((\text{loop}_{13}(p, m, z) \land z \leq 0) \Rightarrow (\text{reset}_{\text{pre}}(p, m) \land \forall p' \cdot \forall m' \cdot (\text{reset}_{\text{sum}}(p, m, p', m') \Rightarrow \varphi_{\text{Exit}})) \)
\]

\[
\varphi_{\text{Entr}} := \forall p \cdot \forall m \cdot \text{drain}_{\text{pre}}(p, m) \Rightarrow (\text{capacity}_{\text{pre}}(p, m) \land \forall z \cdot (\text{capacity}_{\text{sum}}(p, m, z) \Rightarrow \varphi_{\text{Loop}}))
\]

Finally, we construct the verification conditions for \( \text{main} \). Since \( \text{main} \) is the entry-point to \( \mathcal{P} \), then \( \text{main} \) must be safe for all possible inputs. This means that \( \text{main} \) does not require a summary. The conditions are as follows.
As outlined in Sec. 3.3, ToCHC(\(P\)) = \(\varphi_{\text{Main}} \land \varphi_{\text{Cons}} \land \varphi_{\text{Res}} \land \varphi_{\text{Cap}} \land \varphi_{\text{Inc}} \land \varphi_{\text{Dr}}\). Next, ToCHC(\(P\)) is strengthened by the predicate template \(I_{0}(C\text{Inv})\) to obtain CHCSynth(\(P, I_{0}\)) = ToCHC(\(P\)) \land (\(\forall m \cdot \forall p \cdot \bot \Rightarrow C\text{Inv}(m, p)\)). Clearly the term \(\bot \Rightarrow C\text{Inv}(m, p)\) is trivially satisfied. This is because the predicate template \(I_{0}(C\text{Inv})\) is also trivial. In general, this need not be the case. Nonetheless, the term ToCHC(\(P\)) is non-trivial. If ToCHC(\(P\)) is provided to a CHC-solver, then the CHC-solver will return a solution \(\sigma\) containing the following components: expressions \(\sigma(C\text{nter}_{\text{pre}})\), \(\sigma(\text{Reset}_{\text{pre}})\), \(\sigma(C\text{apacity}_{\text{pre}})\), \(\sigma(\text{Increment}_{\text{pre}})\), and \(\sigma(\text{Drain}_{\text{pre}})\), which over-approximate the inputs passed to each function; expressions \(\sigma(C\text{nter}_{\text{sum}})\), \(\sigma(\text{Capacity}_{\text{sum}})\), \(\sigma(\text{Increment}_{\text{sum}})\), and \(\sigma(\text{Drain}_{\text{sum}})\), which over-approximate the return values of each function; an expression \(\sigma(\text{loop}_{13})\) which over-approximates the reachable states of the loop in \(\text{Drain}\); an expression \(\sigma(C\text{Inv})\) which describes a safe implementation for \(C\text{Inv}\). In one solution, \(\sigma(\text{loop}_{13}) = (p \leq m \land (x \neq 0 \Rightarrow 0 < p) \land (x = 0 \Rightarrow 0 = p))\). 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 \(\sigma(C\text{Inv}) = (m > 0 \land p \leq m)\). Clearly \(\sigma(C\text{Inv})\) is an \(F\)-solution since \(\sigma(C\text{Inv})\) is a conjunction of linear inequalities. Then by Thm. 10, \(I_{0} : C\text{Inv} \rightarrow (m > 0 \land p \leq m)\) is a solution to \((P, T, I_{0})\) with \(P[I_{0}]\) 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 \(P\) is given to an IPS-MP solver, then a solution to the IPS-MP problem implies that \(P\) is correct, and a witness to unrealizability implies that \(P\) 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) \(\text{inc}(x)\) increases the variable \(x\) by 1 and increment the program counter; (2) \(\text{dec}(x)\) decreases the variable \(x\) by 1 and increment the program counter; (3) \(\text{jump}(x, i)\) goes to location \(i\) if \(x\) is 0, else increments the program counter; (4) \(\text{halt}()\) halts execution of the program. The halting problem for 2-counter machines is known to be undecidable [49].

\textbf{Theorem 14.} The IPS-MP problem is undecidable for linear integer arithmetic.
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, \( \text{ToCHC}(f) \) relates the class invariant \( \varphi \) to a summary of each method \( f \) in \( P \), and \( f_{\text{pre}} \) 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 \( (P, T) \) such that \( P \in \text{Progs}(\Sigma, V) \) is an open program as in Figure 10a and \( T \) is a theory. A solution to \( (P, T) \) is a \( \varphi \in \text{QFFml}(\Sigma, \{x, y\}) \) such that the following are \( T \)-satisfiable:

\[
\begin{align*}
\psi_{\text{Init}} &:= \forall V \left( \text{Cl}\text{s}_{\text{pre}}(a) \land \text{Cl}\text{s}_{\text{sum}}(a, x, y) \Rightarrow \varphi \right) \\
\psi_{\text{Close1}} &:= \forall V \left( \varphi \land f_{\text{sum}}(x, y, a, x', y') \Rightarrow \varphi' \right) \\
\psi_{\text{Close2}} &:= \forall V \left( \varphi \land g_{\text{sum}}(x, y, a, b, x', y') \Rightarrow \varphi' \right) \\
\psi_{\text{Suff}} &:= \forall V \left( \varphi \Rightarrow \text{func}_{\text{sum}}(x, y, a) \right)
\end{align*}
\]

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

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...
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 invariants6. 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. 177. Note that in Def. 17 ToCHC relates the compositional invariant ψ to the summary of tr, tr pre enforces that tr is summarized, and φ inf := ψ 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 (P, T) such that P ∈ Progs(Σ, V) is an open program as in Figure 11a and T is a theory. A solution to (P, T) is a φ ∈ QFFml(Σ, {1, s, r}) such that the following are T-satisfiable given φ inf := ψ l/r s/i r/l:

ψ init := ∀V (init(l, r, s, r) ⇒ φ)  
ψ close := ∀V (φ ∧ φ inf ∧ tr sum(l, s, r, l′, s′, r′) ⇒ φ′)  
ψ adv := ∀V (φ ⇒ property(l, s, r))  
ψ ins := ∀V (φ ∧ φ inf ∧ tr sum(l, s, r, l′, s′, r′) ⇒ φ inf′)  
ψ sum := ToCHC(P) ∧ ∀V : (φ ∧ φ inf ⇒ 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.
Table 1 Performance of various solvers on IPS-MP benchmarks.

<table>
<thead>
<tr>
<th>Type</th>
<th>Safe</th>
<th>Buggy</th>
<th>Preds</th>
<th>Size</th>
<th>IPS-MP (SEAHORN)</th>
<th>IPS-MP (Eldarica)</th>
<th>HorsSpec</th>
<th>CVC4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>TO</td>
<td></td>
<td></td>
<td>Time TO MEM</td>
<td>Time UN TO N/A</td>
<td>TO N/A</td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td>7</td>
<td>2</td>
<td>107</td>
<td>179 KB</td>
<td>0 14 12 14 4 12 2</td>
<td>7 7 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>6</td>
<td>6</td>
<td>694</td>
<td>6 KB</td>
<td>4 0 10 230 0 10</td>
<td>— 12 0 6 6 0</td>
<td>— 12 0 6 6 0</td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>4</td>
<td>6</td>
<td>535</td>
<td>KB</td>
<td>1 1 0 52 0 5</td>
<td>— 5 0 2 3 0</td>
<td>— 5 0 2 3 0</td>
<td></td>
</tr>
<tr>
<td>Proc</td>
<td>3</td>
<td>3</td>
<td>418</td>
<td>KB</td>
<td>2 0 6 4 0 6</td>
<td>— 6 0 3 3 0</td>
<td>— 6 0 3 3 0</td>
<td></td>
</tr>
<tr>
<td>SC</td>
<td>70</td>
<td>4</td>
<td>181</td>
<td>975 MB</td>
<td>6 8 7 4 7 0</td>
<td>12 9</td>
<td>— — — — — —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>92</td>
<td>29</td>
<td>207</td>
<td>691 117 14</td>
<td>53 12 56</td>
<td>4 45 2</td>
<td>22 29</td>
</tr>
</tbody>
</table>

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

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 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 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 support for IPS-MP. Of these 70 SC benchmarks, 62 are taken from real-world examples used to manage monetary assets [53].

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.
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 [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 \textit{false} are often sufficient but undesirable. As a consequence, many tools aim to synthesize either \textit{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 \textsc{HornSpec} [54] and demonstrate that \textsc{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 \textsc{SeaHorn} [32], \textsc{SemGuS} [42], and \textsc{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\footnote{https://chc-comp.github.io}, \textsc{Spacer} [43] and \textsc{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 \textsc{SeaHorn} verification framework. Our evaluation demonstrates the effectiveness of CHC solvers in solving IPS-MP when compared with general synthesis tools such as \textsc{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.

9 References


\footnote{https://chc-comp.github.io}


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

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

\section{A Syntax and Semantics}

The syntax of \(\text{Progs}(\Sigma,V)\) 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 \(\text{loop}_{ln}\). The result of \(\text{wlp}(S,Q)\) encodes that \(\text{loop}_{ln}\) is a loop invariant for \(S\) which is sufficient to entail \(Q\).

\section{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
\[
\text{wlp}(S_1; S_2, Q) := \text{wlp}(S_1, \text{wlp}(S_2, Q)) \\
\text{wlp(while}_{\text{ln}} (\varphi) \{S\}, Q) := \forall \vec{w} \cdot ((\text{loop}_{\text{ln}}(\vec{w}) \land \varphi) \Rightarrow \text{wlp}(S, \text{loop}_{\text{ln}}(\vec{w}))) \land \forall \vec{w} \cdot ((\text{loop}_{\text{ln}}(\vec{w}) \land \neg \varphi) \Rightarrow Q) \land \text{loop}_{\text{ln}}(\vec{w}) \\
\text{wlp(if } (\varphi) \{S_1\} \text{ else } \{S_2\}, Q) := (\varphi \Rightarrow \text{wlp}(S_1, Q)) \land (\neg \varphi \Rightarrow \text{wlp}(S_2, Q)) \\
\text{wlp}(\vec{y} = f(\vec{e}), Q) := f_{\text{pre}}(\vec{e}) \land \forall \vec{r} \cdot (f_{\text{sum}}(\vec{e}, \vec{r}) \Rightarrow Q[\vec{y}/\vec{r}]) \\
\text{wlp(skip), Q) := Q} \\
\text{wlp(assume}(\varphi), Q) := \varphi \land Q
\]

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

Figure 14 A program that is correct relative to the loop invariant \(2 \cdot i = y \land (i \leq x)\), 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. Therefore, our loop invariant has the shape \(\text{Inv1}(x, i) \land \text{Inv2}(i, y)\). In the IPS-MP encoding, both \(\text{Inv1}\) and \(\text{Inv2}\) 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 \(i \leq x\) to the hole in \(\text{Inv1}\), and the expression \(y = 2 \cdot i\) to the
hole in \(\text{Inv2}\).
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, $\text{ToCHC}(f)$ is used to relate the loop invariant $\varphi$ to the summary of each procedure $f$ in $P$. 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 $(P, T)$ such that $P \in \text{Progs}(\Sigma, \{x_1, x_2\})$ is a problem with the main procedure from Figure 15a (where $\Sigma$ is a first-order signature), and $T$ is a theory. A solution to $(P, T)$ is a $\varphi \in \text{QFFml}(\Sigma, \{x_1, x_2\})$ such that the following are $T$-satisfiable:

\begin{enumerate}[1.]
  \item $\psi_{\text{pre}} := \forall V \cdot \text{wlp}(S_1, \varphi)$, where $S_1$ is the statement before the loop;
  \item $\psi_T := \forall V \cdot (\varphi \land e) \Rightarrow \text{wlp}(S_2, \varphi)$, where $S_2$ is the loop body and $e$ is the loop condition;
  \item $\psi_{\text{post}} := \forall V \cdot (\varphi \land \neg e) \Rightarrow \text{wlp}(S_3, \top)$, where $S_3$ is the statement after the loop;
  \item $\psi_{\text{procs}} := \bigwedge_{f \in \text{Procs}(P)} \text{ToCHC}(f)$.
\end{enumerate}

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

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

\[
\begin{align*}
\text{ToCHC}(P'[\Pi]) &= \text{wlp}(P'(\text{main}), \top) \land \psi_{\text{procs}} \land \forall V \cdot (\text{Inv}(x, y) \leftrightarrow \varphi) \\
\text{wlp}(P'(\text{main}), \top) &= \forall V \cdot \text{wlp}(S_1, \text{Inv}(x, y)) \land \forall V' \cdot (\text{Inv}(x', y') \Rightarrow ((\epsilon' \Rightarrow \text{wlp}(S_2, \text{Inv}(x', y'))) \land (\neg \epsilon' \Rightarrow \text{wlp}(S_3, \top))))
\end{align*}
\]

As $\forall V \cdot \text{Inv}(x, y) \leftrightarrow \varphi$, then $\text{ToCHC}(P'[\Pi])$ is $T$-satisfiable if and only if $\text{ToCHC}(P'[\Pi])$ is $T$-satisfiable after substituting $\varphi$ for Inv. By substituting $\varphi$ for Inv and rewriting $\text{wlp}(P'(\text{main}), \top)$ as a conjunction of CHCs, it follows that $\text{ToCHC}(P'[\Pi]) \Rightarrow \psi_{\text{pre}} \land \psi_T \land \psi_{\text{post}} \land \psi_{\text{procs}}$. Therefore, $\varphi$ is a solution to $(P, T)$. 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 $O(|\text{Locs}| + |PE|)$ since Init iterates over Partial($P$) and calls InitBoolReach, InitBoolReach iterates over Locs and PE, and |Locs| $\geq$ Partial($P$). The loop on line 22 of Algorithm 2 requires $O(|\text{Locs}| \cdot 2^k)$ iterations, since each state is visited at most once, there are at most $2^k$ 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.
void main ( int x1 , int x2) {
  // Code before loop.
  S1;
  // Loop condition and body.
  while (e) { S2; }
  // Code after loop.
  S3;
}

bool PRED_TEMPLATE Inv ( int x1 , int x2) {
  return synth (x1 , x2); }
void main ( int x1 , int x2) {
  S1;
  assert ( Inv (x1 , x2));
  x1 = *; x2 = *;
  assume ( Inv (x1 , x2));
  if (e) {
    S2;
    assert ( Inv (x1 , x2));
    assume ( false );
  }
  S3;
}

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 \texttt{DoAssumes} performs \(O(|FE|)\) operations since \texttt{DoAssumes} iterates over \(FE\) and does \(O(1)\) operations per edge in \(FE\).
- Each call to \texttt{DoAsserts} performs \(O(|AE|)\) operations since \texttt{DoAsserts} iterates over \(AE\) and does \(O(1)\) operations per edge in \(AE\).
- Each call to \texttt{DoProcSum} iterates over \(PE\). However, \texttt{DoProcSum} only processes \((l_{in}, l_{out}) \in PE\) if \(l_{out} = l_{wk}\). Since each function has a single exit location, there is at most one edge in \(PE\) such that \(l_{out} = l_{wk}\). Therefore, each call to \texttt{DoProcSum} performs \(O(|CE|)\) operations, as once \((l_{in}, l_{out})\) is found, \texttt{DoProcSum} does \(O(1)\) operations, iterates over \(CE\), and does \(O(1)\) operations per edge in \(CE\).
- Each call to \texttt{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, \texttt{BoolSynth} terminates within \(O(2^{2k} \cdot |Locs| \cdot |NE \cup CE \cup FE \cup AE|)\) operations.

### E Proof of Theorem 8

**Proof.** From the correctness of \texttt{ComputeBoolReach} (Algorithm 1):
1. \texttt{DoInterproc} encodes rule 3 of Def. 2;
2. \texttt{DoProc} encodes rules 4 and 5 of Def. 2;
3. \texttt{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. \texttt{DoAssumes} encodes rule 5 of Def. 6;
2. \texttt{DoAsserts} encodes rule 3, and queues work for \texttt{DoFunSum};
3. \texttt{DoFunSum} maintains rules 4 and 5 of Def. 6 for items queued in \texttt{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 \((\theta, \sigma, \Pi)\) on line 21. Both \(\theta\) and \(\sigma\) are initialized according to \texttt{ComputeBoolReach}, and follow rule 2 of Def. 2. The initial assignment to \(\Pi\) is \(\Pi_0\). Therefore, when \texttt{Analyze} terminates, \((\theta, \sigma, \Pi)\) is a least partial program summary such that \(\forall p \in \text{Partial}(P) \cdot \Pi(p) \Rightarrow \Pi'(p)\).

### F Proof of Corollary 9

The following proposition was stated informally in Sec. 3.3.
Proposition 21 ([6]). If \((\theta, \sigma)\) is the least summary of a Boolean program \(P\), then \(P\) is correct if and only if \(\sigma(l_\perp) = 1\).

The proof of Cor. 9 follows.

Proof. Assume that \((P, \varnothing, \Pi_0)\) is an IPS-MP problem for a Boolean program \(P\). By Thm. 8, line 31 computes a least partial program summary \((\theta, \sigma, \Pi)\) for \(P\) such that \(\forall p \in \text{Partial}(P) \cdot \Pi_0(p) \Rightarrow \Pi(p)\). Furthermore, by Thm. 7, the call on line 31 always terminates. Then, \(\Pi\) solves the IPS-MP instance if and only if \(P\) is safe. By Prop. 21, \(P[\Pi]\) is safe if and only if \(\theta(l_\perp) = 1\). On line 32, if \(\theta(l_\perp) = 1\), then a solution is returned, else a witness to unrealizability is returned. Therefore, \(\text{BooLSynth}\) decides the IPS-MP problem for Boolean programs.

\[\blacktriangle\]

\section{Proof of Theorem 10}

The following proposition was stated informally in Sec. 3.3.

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

The proof of Thm. 10 follows.

Proof. Let \(\text{CHCBind}(P) := \bigwedge_{p \in \text{Partial}(P)} \forall \vec{x} \cdot (\Pi_0(p) \Rightarrow p(\vec{x}))\). Assume that \(\sigma\) is an \(F\)-solution to \(\text{CHCSynth}(P)\). Since \(\text{CHCBind}(P)\) is a term of \(\text{CHCSynth}(P)\), then \(\forall p \in \text{Partial}(P): |\sigma_\perp| = |\Pi_0(p)\Rightarrow \Pi(p)\). It then follows by induction on the size of \(\text{Partial}(P)\) that \(P[\Pi]\) is correct.

Hypothesis. For some \(k \geq 0\), if \(|\text{Partial}(P)| \leq k\) and \(\text{CHCSynth}(P, \Pi_0)\) has an \(F\)-solution, then \(P\) is correct.

Base Case. If \(|\text{Partial}(P)| = 0\), then \(P\) is correct by Prop. 22.

Inductive Case. Assume that \(|\text{Partial}(P)| = k + 1\), \(\text{CHCSynth}(P)\) has an \(F\)-solution \(\sigma\), and the inductive hypothesis holds up to \(k\). Let \(p \in \text{Partial}(P)\) and \(P' = P[p \leftarrow \sigma(p)]\). By the definition of an interpretation, \(\sigma\) is also a solution to \(\text{ToCHC}(P) \land \text{CHCBind}(P) \land \forall \vec{x} \cdot p(\vec{x}) \Leftrightarrow \sigma(p)\). 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, \(\sigma\) is an \(F\)-solution to \(\text{CHCSynth}(P')\). By hypothesis, \(P'[\Pi]\) is correct. Since \(\Pi(p) = \sigma(p)\), then \(P'[\Pi] = P[p \leftarrow \sigma(p)][\Pi] = P[\Pi]\). Therefore, \(P[\Pi]\) is also correct.

Therefore, \(\Pi\) is a solution to \((P, T, \Pi_0)\).

\[\blacktriangle\]

\section{Proof of Theorem 11}

The following proposition was stated informally in Sec. 3.3.

Proposition 23 ([14]). \(\text{ToCHC}(P)\) is a \(\text{CHC}\) conjunction.

The proof of Thm. 11 follows.

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

\[
\text{ToCHC}(P[\Pi]) = \text{ToCHC}(P) \land \left( \bigwedge_{p \in \text{Partial}(P)} \forall \vec{x} \cdot (\Pi(p) \Leftrightarrow p(\vec{x})) \right)
\]

Then, \(\text{ToCHC}(P) \land \left( \bigwedge_{p \in \text{Partial}(P)} \forall \vec{x} \cdot (\Pi_0(p) \Rightarrow p(\vec{x})) \right)\) has a \(T\)-solution, since \(\Pi\) is a solution to \((P, T, \Pi_0)\) and therefore satisfies \(\forall p \in \text{Partial}(P): |\Pi(p)\Rightarrow \Pi_0(p)\). Then \(\text{CHCSynth}(P, \Pi_0)\) is \(T\)-satisfiable. By contradiction, \((P, T, \Pi_0)\) is unrealizable.

\[\blacktriangle\]
Proof of Theorem 12

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

Proof of Theorem 14

The following proposition was stated informally in Sec. 5.2.

\begin{proposition}[[49]] \(\text{The halting problem is undecidable for 2-counters.}\)

The proof of Thm. 14 follows.

Proof. Assume for the intent of contradiction that IPS-MP is decidable for linear integer arithmetic. Let \(\mathcal{V} = \{x, y, z\}\), \(\Sigma\) 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 \(\text{Progs}(\Sigma, \mathcal{V})\) as follows:
1. The two integer counters are \(x\) and \(y\).
2. The program counter is \(z\).
3. The body of \(P(\text{main})\) is a \texttt{while} loop with loop condition \texttt{true}.
4. The body of the \texttt{while} loop is a sequence of \texttt{if-else} statements that maps each value of \(z\) to an instruction.
5. The instruction \texttt{inc}(x) is: \(x = x + 1;\) \(z = z + 1;\)
6. The instruction \texttt{dec}(x) is: \(x = x - 1;\) \(z = z + 1;\)
7. The instruction \texttt{jump}(x, i) is: \texttt{if} \(x = 0\) \{ \texttt{z++;} \} \texttt{else} \{ \texttt{z++;} \}
8. The instruction \texttt{halt}() is: \texttt{assert(false)};

If \(P \in \text{Progs}(\Sigma, \mathcal{V})\) is a 2-counter machine (following the above encoding), then \(P\) halts if and only if \(P\) violates an assertion. Furthermore, \(|\text{Partial}(P)| = 0\). Let \(\Pi_0\) be the trivial function from \(\text{Partial}(P)\) to \(\text{QFFml}(\Sigma, \mathcal{V})\). Then the IPS-MP problem \(\langle P, \mathcal{T}, \Pi_0 \rangle\) has a solution if and only if \(P\) 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.

Proof of Theorem 16

Proof. Let \(\Pi\) be a solution \(\langle P', \Pi_1, \mathcal{T} \rangle\). This is true if and only if \(P'[\Pi]\) is correct relative to \(\mathcal{T}\). By Prop. 22, this is true if and only if ToCHC\(\langle P'[\Pi]\rangle\) is \(\mathcal{T}\)-satisfiable. By definition

\[
\text{ToCHC}(P'[\Pi]) = \text{wlp}(P'(\text{main}), \top) \land \text{ToCHC}(P) \land \forall \mathcal{V} \cdot (\text{Inv}(x, y) \leftrightarrow \varphi)
\]

Then \(\text{wlp}(P'(\text{main}), \top)\) is \(\forall \mathcal{V} \cdot (\text{Inv}(x, y) \leftrightarrow \varphi)\).

As a direct result, ToCHC\(\langle P'[\Pi]\rangle\) \(\Rightarrow\) \(\text{Inv} \land \text{psi}\text{.}\text{Closure1} \land \text{psi}\text{.}\text{Closure2} \land \text{psi}\text{.}\text{Closure3} \land \text{psi}\text{.}\text{Closure4}\). Therefore, \(\varphi\) is a solution to \(\langle P, \mathcal{T} \rangle\). The other direction is symmetric.
Proof of Theorem 18

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

\[
\text{ToCHC}(P'[\Pi]) = \text{wlp}(P'(\text{main}), T) \land \psi_{\text{Process}} \land \forall V \cdot (\text{Inv}(l, s, r) \Leftrightarrow \varphi)
\]

\[
\text{wlp}(P'(\text{main}), T) = \forall V \cdot ((\text{br} = 0) \Rightarrow \tau_0) \land ((\text{br} \neq 0) \Rightarrow \tau_1),
\]

where $\tau_i$ is the WLP of the $i$-th branch of $P'(\text{main})$. Since br does not appear in $\tau_0$ or $\tau_1$, then $\text{wlp}(P'(\text{main}), T)$ is equisatisfiable with $\text{ToCHC}(P'[\Pi])$ is $T$-satisfiable after substituting $\varphi$ for $\text{Inv}$. Observe that, after substituting $\varphi$ for $\text{Inv}$ and simplifying to CHC conjunctions:

\[
\tau_0 = (\text{Inv}(l, s, r) \land \text{Inv}(r, i, l)) \Rightarrow (\text{tr}_{\text{pre}}(l, s, r) \land (\text{tr}_{\text{sum}}(l, s, r, l', s', r') \Rightarrow (\text{Inv}(l', s', r') \land \text{Inv}(r', i, l'))))
\]

\[
\tau_1 = \psi_{\text{Adequate}}
\]

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

\[
\tau_0 = \psi_{\text{Closure}} \land \psi_{\text{Inf}} \land \forall V \cdot (\varphi \land \varphi_{\text{Inf}} \Rightarrow \text{tr}_{\text{pre}}(l, s, r))
\]

\[
\tau_1 = \psi_{\text{Adequate}}
\]

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