Maximal Specification Synthesis

Aws Albarghouthi
University of Wisconsin–Madison

Isil Dillig
University of Texas at Austin

Arie Gurfinkel [*]
Carnegie Mellon University

Abstract

Many problems in program analysis, verification, and synthesis require inferring specifications of unknown procedures. Motivated by a broad range of applications, we formulate the problem of *maximal specification inference*: Given a postcondition φ and a program P calling a set of unknown procedures $F_1, \ldots, F_n$, what are the most permissive specifications of procedures F_i that ensure correctness of P? In other words, we are looking for the smallest number of assumptions we need to make about the behaviours of F_i in order to prove that P satisfies its postcondition.

To solve this problem, we present a novel approach that utilizes a counterexample-guided inductive synthesis loop and reduces the maximal specification inference problem to *multi-abduction*. We formulate the novel notion of *multi-abduction* as a generalization of classical logical abduction and present an algorithm for solving multi-abduction problems. On the practical side, we evaluate our specification inference technique on a range of benchmarks and demonstrate its ability to synthesize specifications of kernel routines invoked by device drivers.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Specification techniques

Keywords verification; specification; synthesis

1. Introduction

In the world of program analysis, verification, and synthesis, many problems require inferring specifications of unknown procedures. Specifically, given a program P calling procedures $F_1, \ldots, F_n$, we often want to answer the following question:

> *What is the weakest specification φ_i for each callee F_i that ensures correctness of program P?*

In other words, what are the minimal assumptions that we can make about the callees of P and still be able to prove P correct? Here, we view specifications as logical relations over a procedure's parameters and returns; therefore, weakest (maximal) relations imply most permissive specifications. This question, which we refer to as the *maximal specification inference* problem, appears in many different guises. We now survey a broad spectrum of problems that can be viewed through the lens of maximal specification inference:

- ***Verification of open programs*:** Many programs are written against libraries whose source code is unavailable (e.g., proprietary), too large and complex to be verified, or written in a different programming language. To verify such programs, we need to know specifications of these library methods.

  In recent years, there has been a flurry of interest in *automatically* inferring weakest specifications of library code [7, 12, 48, 54]. For instance, in the context of taint-flow analysis for Android applications, Bastani et al. [12] study synthesizing taint-flow specifications of Android library methods from their usage contexts. They advocate presenting these specifications to a "*human auditor for validation,*" with the goal of avoiding the prohibitively expensive analysis of the whole Android ecosystem or the error-prone task of manually writing stubs.
- ***Compositional interprocedural verification*:** Maximal specification inference facilitates *top-down* compositional verification of programs. Starting from the main procedure, we can infer weakest necessary specifications of callees, which can then be verified recursively down the call graph in a modular fashion.
- ***Modular program synthesis*:** Maximal specification inference can also enable modular program synthesis. For instance, given an incomplete program *sketch* with holes (i.e., missing expressions), we can model each hole as a call to an unknown procedure. Once we infer a maximal specification for each hole, we can then use off-the-shelf code synthesis techniques [8, 34, 38, 51] to independently synthesize each unknown expression. Furthermore, if the goal is to fill holes with constant parameters [51], maximal specification inference can logically and succinctly characterize a large or even infinite solution space of possible parameter combinations.
- ***Input-filter synthesis*:** Input-filter generation—a key problem in security—involves synthesizing a procedure that rejects inputs that crash the program or violate certain correctness criteria (see, e.g., [21, 26, 42]). We can cast this problem as maximal specification inference by inserting an unknown procedure at the beginning of the program that reads the input and accepts it or rejects it. By synthesizing a maximal specification for the unknown procedure, we are effectively synthesizing the most permissive input filter—the one that accepts all good inputs and rejects all bad ones.
- ***Infinite-state games and program repair*:** Infinite-state games can be modeled as programs with unknown procedures representing possible player strategies (moves) [14, 15]. Maximal specification inference allows us to determine whether there exists a winning strategy for a player, and, if so, what that strategy is. In addition, following the observations of Jobstmann et al. [37], we can reduce program repair to a game-solving prob-

[*] Copyright 2015 ACM. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002944

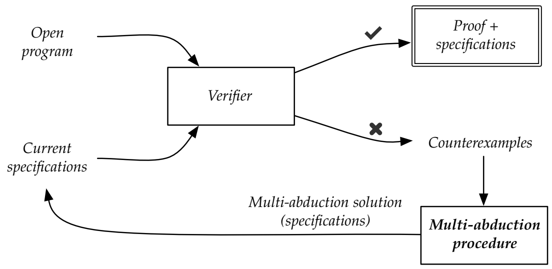


Figure 1: Overview of our approach

lem, which can then be formulated as maximal specification inference. For instance, Beyene et al. [14] and Jobstmann et al. identify a set of "*suspect statements*" in a faulty program. To find a replacement for such statements, they, essentially, abstract them as unknown procedures and synthesize specifications of those procedures that ensure program correctness.

Motivated by such a diverse range of applications, we study the maximal specification inference problem in a general logic-based setting. Specifically, we propose a technique for automatically inferring maximal specifications by reducing the problem to a novel generalization of logical abduction. Given a program P containing calls to procedures $F_1, \ldots, F_n$, we wish to infer a logical specification φ_i for each F_i such that the following conditions are satisfied:

- ***Safety*:** If each procedure F_i satisfies its specification φ_i, then program P is guaranteed to satisfy its desired safety property.
- ***Non-triviality*:** Our technique should not infer the trivial specification *false* for any procedure.
- ***Maximality*:** The specification φ_i inferred for each F_i should be a weakest such specification. That is, we cannot weaken any φ_i without either violating safety or making the specification φ_j for some other procedure F_j stronger.

In our setting, the maximality criterion (Pareto optimality) is very important because we would like to infer the weakest specifications that are needed for guaranteeing safety. Specifications that are stronger than necessary can cause a verifier to miss real bugs or a synthesizer to fail where it should not. For instance, synthesizing an overly restrictive specification of a library routine and then reusing it to verify a client application might cause us to miss bugs.

The key ingredient of our specification inference algorithm is a *counterexample-guided inductive synthesis* (CEGIS) loop with a new logical inference technique called *multi-abduction* at its core (Figure 1). Multi-abduction is a powerful generalization of *standard abduction*: Given a pair of formulas $\mathcal{X}, C$ and an unknown predicate $R(\boldsymbol{x})$ (called an *abducible*), standard abduction infers an interpretation φ of R such that $\varphi \land \mathcal{X} \not\models \mathit{false}$ and $\varphi \land \mathcal{X} \models C$. Intuitively, standard abduction looks for an *explanatory hypothesis* φ that, together with the *known facts* $\mathcal{X}$, is sufficient to imply the *conclusion* C.

Multi-abduction generalizes standard abduction by simultaneously inferring multiple unknowns. Specifically, given a formula

$$\left(\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n_i} R_i(\boldsymbol{x}_{ij}) \land \mathcal{X}\right) \Rightarrow C,$$

multi-abduction computes a map from each unknown predicate R_i to a formula φ_i over variables $\boldsymbol{v_i}$ such that

(*i*) $\bigwedge_{ij} \varphi_i[\boldsymbol{x}_{ij}/\boldsymbol{v}_i] \land \mathcal{X} \not\models \mathit{false}$, and

(*ii*) $\bigwedge_{ij} \varphi_i[\boldsymbol{x}_{ij}/\boldsymbol{v}_i] \land \mathcal{X} \models C$,

where $\varphi[\boldsymbol{x}/\boldsymbol{v}]$ is φ with all occurrences of $\boldsymbol{v}$ replaced by $\boldsymbol{x}$. Observe that multi-abduction generalizes standard abduction in two important ways:

(*i*) The input formula is allowed to contain multiple different unknown predicates $R_1, \ldots, R_n$, and

(*ii*) the same predicate R_i can appear multiple times with different arguments $\boldsymbol{x}_{i1}, \ldots, \boldsymbol{x}_{in_i}$.

```
1 l ← 0
2 while (*)
3    if (p = 1)
4       l ← lock(l)
5    if (p != 1)
6       l ← 0
7    else
8       assert(l = 1)
```

Figure 2: Example program used to illustrate our technique

The problem of inferring unknown method specifications in a bounded program fragment is naturally encoded as multi-abduction: Each unknown specification corresponds to an unknown predicate R_i, and the same R_i can appear multiple times with different arguments at distinct call sites. Further, since we want to infer weakest specifications, we are interested in *maximal solutions* to multi-abduction problems.

As outlined in Figure 1, our approach is based on a CEGIS loop that uses multi-abduction to synthesize specifications from a finite set of bounded control-flow paths through a program P. Given a set of candidate specifications, we then invoke a verifier to check if the specifications ensure correctness of *all* of P's execution paths. If not, new counterexample paths are emitted by the verifier and synthesis is restarted, taking the new program paths into account.

We implemented our approach in an LLVM-based program verifier [6] and used it to synthesize maximal specifications of unknown procedures in a variety of programs. Most interestingly, we show how our approach can efficiently synthesize stubs for kernel routines and initialization code from Windows device drivers used in the software verification competition (SV-COMP) [16].

This paper makes the following key contributions:

- We define the *maximal specification inference* problem and present a *counterexample-guided inductive synthesis* algorithm for solving it.
- We define the *multi-abduction* problem, a generalization of traditional logical abduction, and present an algorithm for finding maximal solutions to multi-abduction problems.
- We show how the specification inference problem for a bounded program fragment can be formulated as multi-abduction.
- We present an implementation of our proposed techniques and demonstrate its applicability by synthesizing maximal specifications in a range of programs, including synthesizing stubs of kernel routines invoked by Windows device drivers.

2. Illustrative Example

In this section, we illustrate our maximal specification inference algorithm by applying it to a simple example.

Consider the program in Figure 2, which is a simplified version of a small `locks` benchmark in the software verification competition (SV-COMP) [16]. Variable `l` takes values in $\{0, 1\}$; the procedure `lock`, called at line 4, is unknown; the symbol $*$ represents non-deterministic choice; and the unassigned variable `p` holds an arbitrary value. Our goal is to synthesize a maximal specification of `lock` such that no program execution violates the assertion at line 8. Since `lock` takes one input argument and returns one value, we are looking for a specification $lock(l_{in}, l_{out})$ that is a relation over the argument and return of `lock`, denoted l_{in} and l_{out}, respectively.

First iteration Our technique begins by setting $lock(l_{in}, l_{out})$ to $true$, the weakest possible specification. It then invokes the verifier to check if this specification is sufficient to prove the assertion. In this case, $true$ is too weak, and the verifier may return the counterexample path $\pi_1 = 1, 2, 3, 4, 5, 8$. This is a counterexample because the specification $true$ implies that `lock` at line 3 may return any value, including 0, which violates the assertion.

Now, our goal is to synthesize a specification of `lock` that ensures that π_1 is correct. To do so, we use π_1 to construct the following multi-abduction query (which, in this case, is just a classical abduction query):

$$(l_{st} = 0 \wedge p = 1 \wedge lock(l_{st}, l_{end})) \Rightarrow l_{end} = 1,$$

The left-hand side of the implication encodes the semantics of the path π_1, where l_{st} denotes the initial (start) value of `l`, l_{end} denotes its value at the assertion, and the call to `lock` is encoded using the *abducible* $lock(l_{st}, l_{end})$. The right-hand side of this implication encodes the assertion (postcondition). Thus, this abduction query is looking for an interpretation of the abducible *lock* that ensures the validity of this implication, and therefore the correctness of path π_1 with respect to the assertion. In this case, we compute the following maximal solution for $lock(l_{in}, l_{out}) \equiv l_{in} = 0 \Rightarrow l_{out} = 1$. This formula specifies that `lock` must return 1 whenever its input is 0; otherwise, `lock` can return an arbitrary value.

Second iteration In the second iteration, we invoke the verifier again, this time checking whether $l_{in} = 0 \Rightarrow l_{out} = 1$ is sufficient to prove correctness of the program. The verifier discovers that this specification is still too weak. As a result, it might return the counterexample $\pi_2 = 1, \langle 2, 3, 4, 5, 8 \rangle^2$, where the superscript indicates that we take the same path through the loop twice. This is a counterexample because, in the second iteration of the loop, the value of `l` is 1 and our specification allows an arbitrary return value for `lock` when the input is 1, thus violating the assertion.

We now create the following new multi-abduction problem:

$$\text{ENCODE}(\{\pi_1, \pi_2\}) \Rightarrow l_{end} = 1,$$

Here, the left-hand side of the implication encodes executions of *both* paths seen so far (i.e., π_1 and π_2), and the right-hand side is the postcondition. Specifically, $\text{ENCODE}(\{\pi_1, \pi_2\})$ corresponds to the following formula:

$$l_{st} = 0 \wedge p = 1 \wedge lock(l_{st}, l_{fst}) \wedge lock(l_{fst}, l_{snd}) \wedge l_{end} \in \{l_{fst}, l_{snd}\}.$$

Observe that the left-hand side of this multi-abduction problem has two occurrences of the unknown predicate *lock* over different arguments, one for each iteration of the loop (*fst* and *snd*). A solution to this multi-abduction problem is an interpretation of the predicate *lock* that makes the implication valid. Therefore, by finding a weakest solution to this multi-abduction problem, we find a maximal specification for `lock` that ensures correctness of the two paths we have examined thus far. In this case, we get the solution $lock(l_{in}, l_{out}) \equiv l_{out} = 1$.

Final iteration Finally, we invoke the verifier with the new specification $l_{out} = 1$, which is sufficient to prove correctness of our program. We thus conclude that a maximal specification for `lock` is one that always returns the value 1, no matter what the input is.

Summary Our approach synthesizes maximal specifications for loop-free program paths by solving multi-abduction problems. Furthermore, even though each iteration of the algorithm considers a finite number of control-flow paths, we can still guarantee the maximality and safety of the inferred specification for the whole program. In the following sections, we formalize this process in detail, present our solutions, and discuss the various challenges and intricacies of developing multi-abduction solvers.

3. Maximal Specifications and Multi-Abduction

We now define the *maximal specification inference* problem and introduce *multi-abduction*. Both in this section and the rest of the paper, we assume that formulas belong to a first-order theory $\mathcal{T}$.

3.1 Maximal safe specifications

Programs A program P is a tuple $(N, E, n_e, n_x, \mathcal{V}, \rho)$, where (N, E) is a directed graph with a special *entry node* n_e and *exit node* n_x. The set $\mathcal{V}$ denotes all program variables, and ρ is a first-order formula that serves as a postcondition for P, denoted $\mathsf{post}(P)$. We assume that each edge $e \in E$ is labelled with a command e^c, which can be one of the following constructs:

- An assignment $v \leftarrow exp$, where $v \in \mathcal{V}$ and *exp* is a program expression over variables $\mathcal{V}$. We assume that there is at most one assignment to each variable (i.e., the program is in SSA form).[1]
- An assumption of the form $assume(b)$, where b is a Boolean expression over program variables $\mathcal{V}$.
- A procedure call $\boldsymbol{x} \leftarrow F(\boldsymbol{y})$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of variables in $\mathcal{V}$ and F is a procedure. We model side effects by allowing procedures to return multiple values.

A procedure F is *unknown* if its source code is not available; we write $\mathsf{unknowns}(P)$ to denote unknown procedures called by P. For the purposes of this paper, we will assume that all callees are unknown, and our goal is to infer safe and permissive specifications for each $F \in \mathsf{unknowns}(P)$. We say that a program P is *closed* if it does not contain calls to unknown procedures; otherwise, it is *open*. Given a closed program P with corresponding postcondition ρ, we write $P \models \rho$ if P satisfies its postcondition in all executions. We also say that program P is *safe* whenever $P \models \rho$.

Maximal safe specifications Given program P with unknown procedures $F_1, \ldots, F_n$, we use the symbol Δ to denote a *specification environment* (or *specification* for short) mapping each procedure F_i to a first-order formula φ_i over vectors of variables $\boldsymbol{\alpha}, \boldsymbol{\beta}$ such that $\boldsymbol{\alpha}$ denotes F_i's arguments and $\boldsymbol{\beta}$ denotes its output. We write $P[\Delta]$ to denote the closed program with every procedure call $\boldsymbol{x} \leftarrow F_i(\boldsymbol{y})$ replaced by $assume(\varphi_i[\boldsymbol{x}/\boldsymbol{\alpha}, \boldsymbol{y}/\boldsymbol{\beta}])$, where $\Delta(F_i) = \varphi_i$. In other words, $P[\Delta]$ is like P but with every unknown procedure F_i behaving according to its specification $\Delta(F_i)$.

DEFINITION 1. **(Safe specification)** *Given a program P, with $\mathsf{post}(P) = \rho$, and a specification environment Δ, we say that Δ is a* safe specification *for P if $P[\Delta] \models \rho$.*

While safe specifications are sufficient for ensuring safety, they may be stronger than necessary. Since we want to synthesize *weakest* safe specifications, we define a partial order on specification environments. We use $dom(\Delta)$ and $range(\Delta)$ to denote the domain and range of Δ.

DEFINITION 2. **(Partial order $\succ$)** *We write $\Delta \succ \Delta'$, and say that Δ is weaker than Δ', if the following hold:*

(i) $dom(\Delta) = dom(\Delta')$,
(ii) $\forall F \in dom(\Delta).\, \Delta(F) \Leftarrow \Delta'(F)$, *and*
(iii) $\exists F \in dom(\Delta).\, \Delta(F) \not\Rightarrow \Delta'(F)$.

We define *maximal safe specifications* using this partial order:

[1] For clarity of presentation, and wlog, we do not include SSA *phi* nodes in the language, as they can be encoded with *assumes* and additional variables.

DEFINITION 3. **(Maximal safe specification)** *Given program P with $\mathsf{post}(P) = \rho$ and a specification environment $\Delta^\star$, we say that $\Delta^\star$ is a* maximal safe specification *for P if*

(i) $P[\Delta^\star] \models \rho$ and
(ii) for all Δ such that $\Delta \succ \Delta^\star$, we have $P[\Delta] \not\models \rho$.

In other words, $\Delta^\star$ is a maximal safe specification for P if we cannot weaken any $\Delta^\star(F)$ while preserving the safety of $P[\Delta^\star]$. Note that there may not be a unique maximal safe specification that is higher than all other safe specifications in $\succ$; instead, there may be a space of incomparable maximal safe specifications.

EXAMPLE 1. *Consider the following program with postcondition $x \geqslant 0 \wedge z > 10$:*

```
x ← F()
y ← G(x)
z ← y + 1
```

A maximal safe specification for this program is

$$\Delta : [F(\beta) \mapsto \beta \geqslant 0,\ G(\alpha, \beta) \mapsto (\alpha \geqslant 0 \Rightarrow \beta > 9)].$$

$\Delta(F)$ specifies that F should always return a positive value. $\Delta(G)$ specifies that G should return a value > 9 whenever its argument is positive; otherwise, it can return an arbitrary value.

3.2 Multi-abduction

Standard abduction We first review standard abduction, which we henceforth refer to as *simple abduction*. Intuitively, simple abduction asks the question, "*What facts do we need to know, in addition to our already-known facts χ, in order to reach conclusion C?"* The following definition formalizes this intuition.

DEFINITION 4. **(Simple abduction)** *Given a formula $R(\boldsymbol{x}) \wedge \chi \Rightarrow C$,* simple abduction *finds a formula φ over variables $\boldsymbol{x}$ such that*

(i) $\varphi \wedge \chi \not\models$ false, and
(ii) $\varphi \wedge \chi \models C$.

In this definition, we refer to $R(\boldsymbol{x})$ as an *abducible*. A *solution* φ to a simple abduction problem is an interpretation of $R(\boldsymbol{x})$ that strengthens the left-hand side of the implication in order to make the implication logically valid. A *maximal* solution to the simple abduction problem is one that is logically weakest. Every solvable simple abduction problem has a unique maximal solution up to logical equivalence.

Given a simple abduction problem $R(\boldsymbol{x}) \wedge \chi \Rightarrow C$, we will assume a procedure $\textsc{abduce}(\chi, C, \boldsymbol{x})$ that computes a maximal solution to the corresponding simple abduction problem. For first-order theories that admit quantifier elimination, one possible implementation of ABDUCE is simply $\textsc{qe}(\forall \overline{\boldsymbol{x}}.\ \chi \Rightarrow C)$, where QE represents a quantifier elimination procedure and $\overline{\boldsymbol{x}}$ denotes all free variables in $\chi \Rightarrow C$ except $\boldsymbol{x}$ [30].

Multi-abduction While simple abduction allows the inference of a single unknown predicate (abducible), it is often necessary to consider multiple abducibles over different vocabularies. For example, in the context of specification inference, different unknown predicates correspond to different procedures, and different occurrences of the same unknown predicate correspond to different calls to the same procedure. Multi-abduction generalizes simple abduction by allowing multiple unknowns as well as different occurrences of the same unknown:

DEFINITION 5. **(Multi-abduction)** *Given a formula*

$$\left(\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n_i} R_i(\boldsymbol{x}_{ij}) \wedge \chi \right) \Rightarrow C,$$

```
1  function MAXSAFESPEC($P$)
2      $\Delta \leftarrow \{F_i \mapsto true \mid F_i \in \mathsf{unknowns}(P)\}$
3      $\Pi \leftarrow \emptyset; \widehat{\Pi} \leftarrow \emptyset$
4      while $true$ do
5          $\theta \leftarrow \textsc{verify}(P[\Delta], \widehat{\Pi})$
6          if $\theta = \emptyset$ then return $\Delta$
7          $\Pi \leftarrow \Pi \cup \theta$
8          $\psi \leftarrow \textsc{encode}(\Pi)$
9          $\Phi \leftarrow \textsc{flatten}(\psi)$
10         $\Delta' \leftarrow \textsc{multiAbduce}(\Phi, \mathsf{post}(P))$
11         if $\Delta' = \mathsf{none}$ then $\widehat{\Pi} \leftarrow \widehat{\Pi} \cup \theta$
12         else $\Delta \leftarrow \Delta'$
```

Algorithm 1: Synthesizing maximal safe specifications

multi-abduction finds a mapping Δ from each R_i to a formula φ_i over variables $\boldsymbol{x}_i$ such that

(i) $\bigwedge_{ij} \varphi_i[\boldsymbol{x}_{ij}/\boldsymbol{x}_i] \wedge \chi \not\models$ false, and
(ii) $\bigwedge_{ij} \varphi_i[\boldsymbol{x}_{ij}/\boldsymbol{x}_i] \wedge \chi \models C$.

In this definition, we call each $R_i(\boldsymbol{x}_{ij})$ an abducible and refer to Δ as a solution to the multi-abduction problem. (For simplicity, we overload notation and use Δ to denote multi-abduction solutions as well as safe specifications.) A *maximal solution* to the multi-abduction problem is one where no φ_i can be weakened. In other words, Δ is a maximal solution to a multi-abduction problem $\mathcal{M}$ if, for any φ_i' that is logically weaker than $\Delta(R_i)$, the mapping $\Delta[R_i \mapsto \varphi_i']$ is *not* a solution to $\mathcal{M}$. Unlike simple abduction problems, multi-abduction problems do not have unique maximal solutions. This is illustrated by the following example:

EXAMPLE 2. *Consider the multi-abduction problem*

$$R_1(x) \wedge R_2(y) \Rightarrow x + y \geqslant 1$$

and two of its solutions

$$\begin{array}{ll} \Delta_1 : & [R_1 \mapsto x \geqslant 0, R_2 \mapsto y \geqslant 1] \\ \Delta_2 : & [R_1 \mapsto x \geqslant 1, R_2 \mapsto y \geqslant\ 0]. \end{array}$$

In this case, both Δ_1 and Δ_2 are maximal and incomparable. □

A special kind of multi-abduction problem is one where all abducibles have unique predicates—that is, each unknown R_i appears exactly once in the formula. Such problems, which we call *linear multi-abduction problems*, have the form $\bigwedge_i R_i(\boldsymbol{x}_i) \wedge \chi \Rightarrow C$. In contrast, we refer to the general case, where the same predicate can appear multiple times with different arguments, as *non-linear multi-abduction problems*.

4. Synthesizing Maximal Safe Specifications

In this section, we present our counterexample-guided inductive synthesis algorithm for learning maximal safe specifications.

High-level description The high-level skeleton of our technique is shown in Algorithm 1; it takes a program P and returns a maximal safe specification. The algorithm initially assumes that each $F \in \mathsf{unknowns}(P)$ has specification $true$ (line 2). While such a specification Δ is very permissive, it is unlikely to be safe. Hence, the algorithm goes through a refinement loop (lines 4–12) where each specification in Δ is iteratively modified until $P[\Delta]$ is proven safe.

Going into more detail, line 5 of Algorithm 1 attempts to verify program P (with respect to its postcondition $\mathsf{post}(P)$) using current specifications Δ. For this purpose, we assume a procedure VERIFY that is capable of generating one or more counterexamples (as program paths) if the verification task fails. VERIFY also takes a set of paths $\widehat{\Pi}$ and ensures that the returned counterexamples are not in $\widehat{\Pi}$, i.e., $\theta \cap \widehat{\Pi} = \emptyset$. Since many existing verification algorithms are capable of generating counterexamples [6, 10, 17, 44], we do not describe the VERIFY procedure in detail.

Now, suppose the verification task at line 5 fails with a set θ of counterexamples that falsify $\mathsf{post}(P)$. In this case, we add θ to the set Π of all counterexamples encountered so far and try to infer a specification environment Δ that is sufficient to rule out all counterexamples in Π.

To infer such a specification Δ, one possibility is to solve a *set* of simultaneous multi-abduction problems. That is, for each counterexample path $\pi \in \Pi$, we generate a multi-abduction problem where the abducibles correspond to procedure calls in π, and we then solve all of these multi-abduction problems *simultaneously*. Unfortunately, there are two main problems with this approach: First, it requires us to devise a technique for solving not just one multi-abduction problem, but a set of them simultaneously. Second, if we represent the set of counterexamples Π symbolically as an SMT formula ψ, generating one multi-abduction problem *per counterexample* would require us to convert ψ to DNF, a possibly exponential operation.

Encoding counterexamples In order to side-step explicit counterexample enumeration and simultaneous multi-abduction, we present a technique that encodes a set of counterexamples into a single multi-abduction problem. Specifically, we think of Π as a loop-free program P_Π and use a procedure called ENCODE to represent all executions of P_Π as a formula ψ (a standard, BMC-like encoding of loop-free programs [5, 11, 24]). Given loop-free program P_Π with nodes N and edges E, ENCODE generates the following formula:

$$c_{n_e} \wedge \bigwedge_{u \in N} \Big(c_u \Rightarrow \bigvee_{e=(u,v) \in E} (c_v \ \wedge \ [\![e^c]\!]) \Big)$$

Here, c_u is an auxiliary Boolean variable indicating that control is at program location u. The formula c_{n_e} indicates that the program begins execution at the entry node n_e. Now, if we reach node u, we must also reach one of its successors v and execute the command e^c associated with edge $e = (u, v)$. Hence, the formula $c_v \wedge [\![e^c]\!]$ denotes the transition from u to v, where $[\![e^c]\!]$ is defined as follows:

$$[\![x \leftarrow e]\!] = \ (x = e) \qquad [\![assume(b)]\!] = \ b$$

$$[\![\boldsymbol{x} \leftarrow F_i(\boldsymbol{y})]\!] = \ R_i(\boldsymbol{x}, \boldsymbol{y})$$

Our encoding of assignments assumes that the program has been converted to SSA form, as we do not explicitly rename variables. Also, since we assume that all callees F_i are unknown, we introduce a predicate R_i that represents the unknown specification of F_i.

Going back to Algorithm 1, let ψ be a formula encoding all counterexamples Π encountered so far. To ensure that each $\pi \in \Pi$ is ruled out in the next iteration, we must find an interpretation of all predicates R_i under which the formula $\psi \Rightarrow \mathsf{post}(P)$ is logically valid. Observe that this is almost a multi-abduction problem, but not quite, as the abducibles R_i can appear under disjunctions in ψ, whereas multi-abduction requires formula ψ to be of the form $\bigwedge_{ij} R_i(\boldsymbol{x}_{ij}) \wedge \chi$ (i.e., all abducibles appear at the outermost conjunction). Fortunately, given a formula ψ where all unknowns appear positively (i.e., under an even number of negations), we can always transform ψ to the syntactic form $\bigwedge_{ij} R_i(\boldsymbol{x}_{ij}) \wedge \chi$ using a procedure we call *flattening*.

Multi-abduction problem flattening Given an arbitrary formula φ where unknown predicates appear only positively, the procedure FLATTEN, given in Algorithm 2, shows how to convert φ to a multi-abduction problem. Specifically, procedure FLATTEN takes as input a formula φ, where all abducibles appear only positively in φ, and returns a formula of the form $\bigwedge_{ij} R_i(\boldsymbol{x}_{ij}) \wedge \chi$, where all abducibles have been *pulled out*.

```
1 function FLATTEN($\varphi$)
2     $\varphi^\star \leftarrow \varphi$; $\psi \leftarrow true$
3     for all $R(\boldsymbol{x}) \in \mathsf{abds}(\varphi^\star)$ do
4         $\boldsymbol{x}_i \leftarrow \mathsf{fresh}(\boldsymbol{x})$
5         $\varphi^\star \leftarrow \varphi^\star[(\boldsymbol{x} = \boldsymbol{x}_i)/R(\boldsymbol{x})]$
6         $\psi \leftarrow \psi \wedge R(\boldsymbol{x}_i)$
7     return $\psi \wedge \varphi^\star$
```

Algorithm 2: Extracting abducibles to the outer level

The main idea behind FLATTEN is to replace every abducible $R(\boldsymbol{x})$ in φ with the formula $\boldsymbol{x} = \boldsymbol{x}_i$ where $\boldsymbol{x}_i$ is a fresh set of variables. After this transformation has been applied to φ for every $R_i(\boldsymbol{x}_{ij})$, we obtain a formula $\varphi^\star$ that does not contain any unknown predicates (line 5). To obtain the final *flattened* formula, we then generate

$$\varphi^\star \wedge \bigwedge_{R(\boldsymbol{x}) \in \mathsf{abd}(\varphi)} R(\boldsymbol{x}_i),$$

where $\boldsymbol{x}_i$ is the fresh set of variables associated with abducible $R(\boldsymbol{x})$ and $\mathsf{abd}(\varphi)$ is the set of all abducibles appearing in φ.

EXAMPLE 3. *Consider the formula $\varphi: \ G(a, b) \vee a = 3 \vee G(c, d)$. Applying* FLATTEN *to φ yields the formula: $\psi \wedge \varphi^\star$, where*

$$\begin{array}{rl} \psi: & G(x_1, x_2) \wedge G(x_3, x_4) \\ \varphi^\star: & (x_1 = a \wedge x_2 = b) \vee a = 3 \vee (x_3 = c \wedge x_4 = d) \end{array} \qquad \Box$$

The following theorem states the correctness of FLATTEN by showing that it preserves non-trivial, maximal, safe solutions, where a non-trivial solution does not set any abducible to $false$.

THEOREM 1 (Correctness of FLATTEN). *Let φ be a formula containing abducibles appearing only positively. Let C be an abducible-free formula. Then,*

$$\varphi[\Delta] \models C \ \textit{ iff } \ \text{FLATTEN}(\varphi)[\Delta] \models C,$$

where Δ is a non-trivial assignment of abducibles to formulas.

Correctness Going back to Algorithm 1, the formula Φ obtained at line 9 encodes all counterexamples Π in a form where all unknowns have been pulled out and has the syntactic form $\bigwedge_{ij} R_i(\boldsymbol{x}_{ij}) \wedge \chi$. Since $\Phi \Rightarrow \mathsf{post}(P)$ now corresponds to a multi-abduction problem, we can compute a maximal solution for the unknown predicates of Φ using the algorithm described in Section 5.

Now, if the call to MULTIABDUCE at line 10 returns a solution, we obtain a new specification environment Δ and continue our refinement loop. However, if the multi-abduction problem does not have a solution (i.e., $\Delta' = \mathsf{none}$ at line 11), this means that the multi-abduction problem only has solutions that make every path in Π infeasible. Hence, we add the most recent counterexamples θ to set $\widehat{\Pi}$; this just ensures that VERIFY does not return a path in $\widehat{\Pi}$ in the next iteration. $\widehat{\Pi}$ is maintained simply to ensure that we do not synthesize a specification that makes all program paths infeasible.

The following theorem states correctness of MAXSAFESPEC. Given the undecidability of the problem, it is not guaranteed to terminate in general. However, for programs with variables over finite domains (e.g., Boolean programs), it will eventually terminate.

THEOREM 2 (Correctness of MAXSAFESPEC). *If* MAXSAFESPEC *terminates on program P, the computed specification environment Δ is a safe, maximal, non-trivial specification for P.*

5. Multi-abduction Algorithm

In this section, we present an algorithm for computing maximal solutions to multi-abduction problems.

```
 1 function MULTIABDUCELINEAR($\mathcal{X}, C, \mathcal{A}$)
 2     $\psi \leftarrow$ ABDUCE($\mathcal{X}, C, \mathsf{vars}(\mathcal{A})$)
 3     if $\psi = \mathsf{none}$ then return none
 4     return CARTDECOMP($\mathcal{X}, \psi, \mathcal{A}$)

 5 function CARTDECOMP($\mathcal{X}, \psi, \mathcal{A}$)
 6     $\Delta \leftarrow$ INITSOLN($\mathcal{X}, \psi, \mathcal{A}$)
 7     for all $R_i(\boldsymbol{x}_i) \in \mathcal{A}$ do
 8         $\Delta(R_i) \leftarrow$ ABDUCE($\bigwedge_{j \neq i} \Delta(R_j), \psi, \boldsymbol{x}_i$)
 9     return $\Delta$

10 function INITSOLN($\mathcal{X}, \psi, \mathcal{A}$)
11     $M \leftarrow \mathsf{model}(\mathcal{X} \wedge \psi)$
12     for all $R_i(\boldsymbol{x}_i) \in \mathcal{A}$ do
13         $\Delta(R_i) \leftarrow (\boldsymbol{x}_i = M(\boldsymbol{x}_i))$
14     return $\Delta$
```

Algorithm 3: Solving linear multi-abduction problems

5.1 Solving linear multi-abduction problems

Before we present our multi-abduction algorithm in its full generality, we will first present an algorithm for solving linear multi-abduction problems. Recall from Section 3 that a linear multi-abduction problem is of the form $\bigwedge_i R_i(\boldsymbol{x}_i) \wedge \mathcal{X} \Rightarrow C$ (i.e., each unknown appears only once in the formula). While the full algorithm is a natural generalization of this one, the algorithm we present in this section is useful for gaining intuition.

Algorithm 3 summarizes our approach for solving linear multi-abduction problems. The procedure MULTIABDUCELINEAR takes as input formulas $\mathcal{X}$ and C, which correspond to the unknown-free formulas on the left- and right-hand sides of the implication in Definition 5, respectively. MULTIABDUCELINEAR also takes as input a set $\mathcal{A}$ of abducibles of the form $R_i(\boldsymbol{x}_i)$ and returns a maximal solution Δ to the problem. We use $\mathsf{vars}(\mathcal{A})$ to denote the set of all arguments of all abducibles in $\mathcal{A}$. The high-level structure of our multi-abduction algorithm consists of two steps:

1. ***Simple abduction*:** First, we solve the simple abduction problem defined by $R^\star(\boldsymbol{x}) \wedge \mathcal{X} \Rightarrow C$, where $\boldsymbol{x}$ denotes all variables appearing in the abducibles. In other words, we combine all abducibles into one abducible $R^\star$ and find a monolithic formula that, informally speaking, *contains* all solutions of the multi-abduction problem. We assume that the ABDUCE procedure returns none if the simple abduction problem has no solution.
2. ***Cartesian decomposition*:** Given a solution ψ to the simple abduction problem from step (1), we decompose ψ into a set of formulas $\varphi_1, \ldots, \varphi_n$ such that

   (*i*) each φ_i is *only* over $\boldsymbol{x}_i$,

   (*ii*) $\bigwedge_i \varphi_i \Rightarrow \psi$, and

   (*iii*) the set of formulas $\varphi_1, \ldots, \varphi_n$ is as weak as possible.

   We refer to the decomposition of formula ψ into such a set $\varphi_1, \ldots, \varphi_n$ as *Cartesian decomposition*—intuitively, we are decomposing ψ into a Cartesian product of the formulas φ_i.

Since we already know how to solve simple abduction problems (Section 3.2), the key part of our multi-abduction algorithm is the Cartesian decomposition function (CARTDECOMP) of Algorithm 3 (line 5). This procedure first finds an initial (non-maximal) solution $\Delta : [R_i \mapsto \varphi_i]$ (line 6) and then iteratively weakens this solution (lines 7-8) until we obtain a maximal one.

The task of finding an initial solution Δ is performed by the INITSOLN procedure. Specifically, INITSOLN finds an interpretation of each R_i of the form $\boldsymbol{x}_i = \boldsymbol{m}_i$, where $\boldsymbol{m}_i$ is a vector of constants. Hence, our initial solution corresponds to a concrete assignment or *point*. To obtain such a solution, we simply get a model M of the formula $\mathcal{X} \wedge \psi$ and assign each R_i to $\boldsymbol{x}_i = M(\boldsymbol{x}_i)$, where $M(\boldsymbol{x}_i)$ is the vector of assignments to variables $\boldsymbol{x}_i$ in the model M. Since this interpretation is consistent with $\mathcal{X}$, and since $\bigwedge_i \Delta(R_i) \Rightarrow \psi$, we know that Δ is a solution to the multi-abduction problem, albeit not necessarily a maximal one.

Now, the loop in lines 7-8 of the CARTDECOMP function iteratively weakens the solution returned by INITSOLN, with the goal of making it maximal. To weaken the solution for each R_i, we *fix* the solution for all the other R_j's and weaken R_i as much as possible by solving the following simple abduction problem:

$$\Big(R_i(\boldsymbol{x}_i) \wedge \bigwedge_{i \neq j} \Delta(R_j)\Big) \Rightarrow \psi.$$

In other words, we use the existing solution given by Δ for all other R_j's and infer the weakest R_i that still implies ψ. This strategy ensures that Δ remains a valid solution to our multi-abduction problem in every iteration.

EXAMPLE 4. *Consider the following multi-abduction problem:*

$$R_1(x) \wedge R_2(y) \wedge x \geqslant 0 \Rightarrow (x + y \geqslant 1 \vee z \geqslant 0).$$

The algorithm starts by solving the following simple abduction problem:

$$R^\star(x, y) \wedge x \geqslant 0 \Rightarrow (x + y \geqslant 1 \vee z \geqslant 0).$$

This problem has maximal solution $R^\star \mapsto (x \geqslant 0 \Rightarrow x + y \geqslant 1)$. Next, using INITSOLN*, we get a model of this formula that is consistent with $x \geqslant 0$, say $x = 0$ and $y = 1$. Hence, the initial solution is $R_1 \mapsto x = 0, R_2 \mapsto y = 1$. Now, we fix R_2 to $y = 1$ and try to weaken R_1 by solving the simple abduction problem:*

$$(R_1(x) \wedge y = 1 \wedge x \geqslant 0) \Rightarrow x + y \geqslant 1.$$

The answer to this problem is $R_1 \mapsto$ true; hence, our current solution becomes $R_1 \mapsto \mathit{true}, R_2 \mapsto y = 1$. Now, we try to weaken R_2 by fixing R_1 and solving the simple abduction problem:

$$(\mathit{true} \wedge R_2(y) \wedge x \geqslant 0) \Rightarrow x + y \geqslant 1.$$

Since ABDUCE *yields $y \geqslant 1$, the final solution is*

$$R_1 \mapsto \mathit{true}, R_2 \mapsto y \geqslant 1.$$

The following theorem states soundness and completeness of our multi-abduction procedure.

THEOREM 3 (Correctness of MULTIABDUCELINEAR). *If the multi-abduction problem is solvable, Algorithm 3 terminates with a maximal solution; otherwise, it returns* none.

5.2 Solving non-linear multi-abduction problems

We now consider our algorithm for solving non-linear multi-abduction problems (see Algorithm 4). The general structure of this algorithm is similar to Algorithm 3 in that we first solve a simple abduction problem and then apply Cartesian decomposition. However, since we must assign the *same interpretation* to two different abducibles $R_i(\boldsymbol{x}_{ij})$ and $R_i(\boldsymbol{x}_{ik})$, the CARTDECOMP function of Algorithm 4 is somewhat more involved.

Like in the linear case, the main procedure MULTIABDUCE takes as input formulas $\mathcal{X}$ and C and a set of abducibles $\mathcal{A}$ of the form $R_i(\boldsymbol{x}_{ij})$—i.e., $\mathcal{A}$ potentially contains several occurrences of the same unknown R_i. We use $\mathsf{preds}(\mathcal{A})$ to denote the set of unique predicates $R_1, \ldots, R_n$ appearing in $\mathcal{A}$.

Cartesian decomposition As in Algorithm 3, the CARTDECOMP procedure first computes an initial non-maximal solution Δ for each abducible R_i (line 6) and then iteratively weakens Δ to obtain a maximal solution (loop at line 7). The difference here is in the way CARTDECOMP treats repeated occurrences of the same unknown. The weakening loop at line 7 works in two steps:

1. In line 8, we group all occurrences of each R_i as one monolithic abducible $R_i^\star$ over the variables $\boldsymbol{x}_{i1}, \ldots, \boldsymbol{x}_{in_i}$, and computes a maximal solution for $R_i^\star$. This is done using the simple abduction query:

   $$\Big(R_i^\star(\boldsymbol{x}_{i1}, \ldots, \boldsymbol{x}_{in_i}) \wedge \bigwedge_{k\neq i} \bigwedge_j \Delta(R_k)[\boldsymbol{x}_{kj}/\boldsymbol{x}_j]\Big) \Rightarrow \psi.$$

   The computed solution ψ_i at line 8 is the weakest formula over variables $\boldsymbol{x}_{i1}, \ldots, \boldsymbol{x}_{in_i}$ that makes the above implication valid.

2. At this point, we need to decompose ψ_i into n_i formulas, one for each occurrence of R_i. These formulas also have to be equivalent, as per definition of the multi-abduction problem. More formally, we would like to find a weakest formula φ_i over the fresh vector of variables $\boldsymbol{x}_i$ such that: $\bigwedge_j \varphi_i[\boldsymbol{x}_{ij}/\boldsymbol{x}_i] \Rightarrow \psi_i$. We call this process *isomorphic decomposition* (line 9), as it decomposes the formula ψ_i into a conjunction of formulas $\varphi_i[\boldsymbol{x}_{ij}/\boldsymbol{x}_i]$ that are equivalent modulo variable renaming.

Initial solution Since the main difference of Algorithm 4 from MULTIABDUCELINEAR lies in isomorphic decomposition and initial solution inference, we now turn our attention to lines 11-27 of the algorithm. Let us first consider the INITSOLN procedure for finding an initial non-maximal interpretation for each unknown R_i. First of all, if the multi-abduction problem is to have a solution, then there must be some interpretation of R_i of the form $\bigvee_j \boldsymbol{x}_i = \boldsymbol{m}_{ij}$, where each $\boldsymbol{m}_{ij}$ is a vector of constants. Note that we do not require these $\boldsymbol{m}_{ij}$'s to be distinct; hence, even if the only interpretation of R_i is a single *point*, we can still find such a solution.

Now, to find a solution of this form, we need to find values of $\boldsymbol{m}_{ij}$ under which the formula

$$\left(\bigwedge_{ij} \bigvee_k \boldsymbol{x}_{ij} = \boldsymbol{m}_{ik}\right) \Rightarrow \psi$$

is valid. Specifically, we let $\boldsymbol{m}_{ij}$ be fresh symbolic constants and ask for a model M of the formula shown in line 13, where $\mathcal{A}(R_i)$ denotes all abducibles containing unknown R_i and $\mathsf{vars}(\psi)$ is the set of free variables in ψ. Then, we obtain an initial solution for $\Delta(R_i)$ as $\bigvee_j \boldsymbol{x}_i = M(\boldsymbol{m}_{ij})$. The intuition here is as follows: In the linear case, INITSOLN initializes every abducible R_i with an assignment $\boldsymbol{x}_i = \boldsymbol{m}_i$. In the non-linear case, we do the same, but we need to make sure that if we initialize one occurrence of R_i with some assignment A, then every other occurrence should also have that assignment. This is captured by the formula in line 12.

Isomorphic decomposition Let us now consider the ISODECOMP procedure in line 17. Given a formula ψ and a set of abducibles $\mathcal{R} = \{R_i(\boldsymbol{x}_{i1}), \ldots, R_i(\boldsymbol{x}_{in_i})\}$, recall that the goal of isomorphic decomposition is to find a weakest formula φ such that

$$\bigwedge_j \varphi[\boldsymbol{x}_{ij}/\boldsymbol{x}_i] \Rightarrow \psi \qquad (1)$$

is valid. To find such a formula φ, ISODECOMP first initializes φ to the current solution of R_i given by Δ. Note that, by construction of the initial solution and the maximality guarantee of ABDUCE, $\Delta(R_i)$ always satisfies Formula 1. Now, starting from this initial solution at line 18, ISODECOMP iteratively weakens φ until it can no longer be weakened (lines 19-27).

In each iteration of the while loop, we weaken our current solution φ by finding a *set* Θ of assignments $\boldsymbol{x}_i = \boldsymbol{m}_{ij}$ such that (*i*) $\boldsymbol{x}_i = \boldsymbol{m}_{ij}$ currently does not satisfy φ, and (*ii*) adding Θ to the satisfying assignments of φ does not violate Formula 1. If such an assignment does not exist, this means φ is already the weakest solution that makes Formula 1 valid; hence, we return φ (line 22). Otherwise, we weaken our current solution φ by adding models Θ (loop at line 25).

```
1  function MULTIABDUCE($\mathcal{X}, C, \mathcal{A}$)
2      $\psi \leftarrow$ ABDUCE($\mathcal{X}, C, \mathsf{vars}(\mathcal{A})$)
3      if $\psi = \mathsf{none}$ then return none
4      return CARTDECOMP($\mathcal{X}, \psi, \mathcal{A}$)

5  function CARTDECOMP($\mathcal{X}, \psi, \mathcal{A}$)
6      $\Delta \leftarrow$ INITSOLN($\mathcal{X}, \psi, \mathcal{A}$)
7      for all $R_i \in \mathsf{preds}(\mathcal{A})$ do
8          $\psi_i \leftarrow$ ABDUCE($\bigwedge_{k\neq i} \bigwedge_j \Delta(R_k)[\boldsymbol{x}_{kj}/\boldsymbol{x}_k], \psi, \boldsymbol{x}_i$)
9          $\Delta(R_i) \leftarrow$ ISODECOMP($\psi_i, \mathcal{A}(R_i), \Delta$)
10     return $\Delta$

11 function INITSOLN($\mathcal{X}, \psi, \mathcal{A}$)
12     $\psi_M \leftarrow \forall\, \mathsf{vars}(\psi).\ \left(\bigwedge_{R_i(\boldsymbol{x}_{ij})\in\mathcal{A}} \bigvee_{k=1}^{|\mathcal{A}(R_i)|} \boldsymbol{x}_{ij} = \boldsymbol{m}_{ik}\right) \Rightarrow \psi$
13     $M \leftarrow \mathsf{model}\big(\mathcal{X}[\boldsymbol{m}_{ij}/\boldsymbol{x}_{ij}] \wedge \psi_M\big)$
14     for all $R_i \in \mathsf{preds}(\mathcal{A})$ do
15         $\Delta(R_i) \leftarrow \bigvee_j \boldsymbol{x}_i = M(\boldsymbol{m}_{ij})$
16     return $\Delta$

17 function ISODECOMP($\psi, \mathcal{R}, \Delta$)
18     $\varphi \leftarrow \Delta(\mathsf{preds}(\mathcal{R}))$
19     while $true$ do
20         $\psi_M \leftarrow \forall\, \mathsf{vars}(\psi).\ \left(\bigwedge_{R(\boldsymbol{x}_{ij})\in\mathcal{R}} \bigvee_{k=1}^{|\mathcal{R}|} \boldsymbol{x}_{ij} = \boldsymbol{m}_{ik}\right) \Rightarrow \psi$
21         $M \leftarrow \mathsf{model}\left(\bigwedge_{j=1}^{|\mathcal{R}|} \neg\varphi[\boldsymbol{m}_{ij}/\boldsymbol{x}_i] \wedge \psi_M\right)$
22         if $M = \emptyset$ then return $\varphi$
23         $\varphi \leftarrow \varphi \vee \bigvee_j \boldsymbol{x}_i = M(\boldsymbol{m}_{ij})$
24         $\varphi_j \leftarrow \varphi[\boldsymbol{x}_{ij}/\boldsymbol{x}_i]$ for each $j$
25         for all $R(\boldsymbol{x}_{ij}) \in \mathcal{R}$ do
26             $\varphi_j \leftarrow$ ABDUCE($\bigwedge_{j\neq k} \varphi_k, \psi, \boldsymbol{x}_{ij}$)
27         $\varphi \leftarrow \bigwedge_j \varphi_j[\boldsymbol{x}_i/\boldsymbol{x}_{ij}]$
```

Algorithm 4: Solving non-linear multi-abduction problems

Lines 24-27 of the algorithm are heuristics to accelerate convergence. In particular, since the ISODECOMP algorithm (as described so far) adds a finite, and typically small, set of models in each iteration, we try to further weaken φ using simple abduction. Specifically, at line 26, we ask, "*How much can we weaken $R_i(\boldsymbol{x}_{ij})$ while fixing the other $R_i(\boldsymbol{x}_{ik})$'s and maintaining the validity of Formula 1?*" Hence, the formula φ_j obtained at line 27 has the property that $\varphi \Rightarrow \varphi_j[\boldsymbol{x}_i/\boldsymbol{x}_{ij}]$. Finally, since all abducibles $R_i(\boldsymbol{x}_{ij})$ must have the same interpretation modulo renaming, we obtain a new solution φ as $\bigwedge_j \varphi_j[\boldsymbol{x}_i/\boldsymbol{x}_{ij}]$ (line 27). Note that the formula φ at line 27 is at least as weak as the version of φ at line 23, although it may not be *strictly* weaker.

EXAMPLE 5. *Consider the multi-abduction problem*

$$R(x) \wedge R(y) \Rightarrow x + y \geqslant 2$$

*in the first-order theory of linear integer arithmetic (*LIA*). The algorithm calls* CARTDECOMP *on the formula $x + y \geqslant 2$, and, regardless of what* INITSOLN *returns, it calls* ISODECOMP *with $\psi = x + y \geqslant 2$ and abducibles $\mathcal{R} = \{R(x), R(y)\}$. Now, assuming* INITSOLN *sets $R(\alpha) \equiv \alpha = 2 \vee \alpha = 3$, we weaken the solution φ by computing a new model: Suppose that φ at line 23 is now assigned to $\alpha = 2 \vee \alpha = 3 \vee \alpha = 4$. In the loop at line 25, we first use simple abduction to weaken $R(x)$, which yields $x \geqslant 0$. In the next iteration, we weaken $R(y)$ to $y \geqslant 2$; hence* *the new φ becomes $\alpha \geqslant 2$ at line 27. In the next iteration of the while loop, we ask for a new model of x and y and obtain $x = 1$, $y = 1$. Hence, the new φ at line 23 becomes $\alpha \geqslant 2 \vee \alpha = 1$. When we use simple abduction to weaken $R(x)$ in line 26, we now obtain $R(x) \equiv x \geqslant 1$. Similarly, when we weaken $R(y)$ using $R(x) \equiv x \geqslant 1$, we obtain $y \geqslant 1$ for $R(y)$. Hence, the final solution is $R(\alpha) \equiv \alpha \geqslant 1$.* □

Correctness and termination The following theorem states soundness of the MULTIABDUCE algorithm.

THEOREM 4 (Correctness of MULTIABDUCE). *If* MULTIABDUCE *returns solution Δ, then Δ is a maximal solution; if it returns* none*, then there is no solution to the multi-abduction problem.*

While the MULTIABDUCE procedure of Algorithm 4 always terminates for logical theories with finite ascending chains of implications (e.g., propositional logic), it does not have termination guarantees otherwise. This is expected because multi-abduction problems in some theories are not guaranteed to have maximal solutions that are finitely expressible in that theory. For example, consider the multi-abduction problem $R(x) \wedge R(y) \Rightarrow y \neq x + 1$ in linear real arithmetic. A maximal solution for $R(\alpha)$ is $\bigvee_{k \in \mathbb{Z}} (2k < \alpha \leqslant 2k + 1)$, which is not finitely expressible in this theory and requires integral constraints. We formalize lack of maximal solutions for non-linear multi-abduction in the following theorem.

THEOREM 5 (Maximality in FOL theories). *There are first-order theories for which there exists non-linear multi-abduction problems that have no maximal solutions.*

6. Implementation and Extensions

6.1 Implementation

We implemented a prototype of our maximal specification synthesis algorithm (MAXSAFESPEC) and its multi-abduction core as an extension of UFO [6], an open-source, LLVM-based [2] verification framework for C code. We extended UFO's front end to enable procedure calls with multiple returns, just as in our program model from Section 3. This model allows us to handle procedures that can modify global state.

To compute inductive invariants, UFO unrolls a program's CFG into a DAG and uses DAG *interpolants* [4] to hypothesize an inductive invariant in case all paths in the unrolling are safe. Our implementation utilizes the DAG-shaped counterexamples emitted by UFO in order to drive the specification synthesis loop. Of course, we could instrument UFO to emit a single counterexample path at a time—instead of a DAG of counterexamples. However, we found that this process does not scale at all as soon as we go beyond very simple programs, due to the large number of paths that need to be examined in order to arrive at a maximal safe specification.

Our multi-abduction algorithm implementation uses the Z3 SMT solver [28] for satisfiability checking and quantifier elimination. Programs are encoded in linear integer arithmetic (QFLIA).

6.2 Optimizations

To produce an efficient implementation of our specification synthesis algorithm, we identified a number of key optimizations.

Aggressive formula preprocessing Unsurprisingly, the most expensive operation in our procedure is the universal quantifier elimination step needed for simple abduction. In our implementation context, we have to apply quantifier elimination to large formulas over linear integer arithmetic (also known as Presburger arithmetic). It is well known that the worst case space/time bound for this problem is super exponential. In addition to this theoretical intractability, quantifier elimination has not received the same attention—in terms of algorithmic and engineering advances—as did satisfiability. To make matters even worse, we are dealing with large formulas encoding CFG unrollings, whereas typical uses of quantifier elimination are restricted to relatively small formulas (e.g., encoding sets of states in some numerical abstract domain).

In our initial attempts, we implemented a simple abduction procedure that directly passes the formula to Z3's quantifier elimination procedure [19]—a combination of Cooper's algorithm [25] and the Omega test [46]. This turned out to be very impractical. As such, we devised a two-step preprocessing phase that (*i*) aggressively simplifies the formula by identifying and removing redundant subformulas and (*ii*) attempts to eliminate *easy* variables by examining the formula syntactically. After performing these two steps, we fall back on Z3 to eliminate the remaining variables from the simplified formula.

First, to simplify a formula φ into an equivalent formula φ_s, we exploit UNSAT cores. Specifically, we ask the SMT solver, "*Which literals in φ can we replace with true without weakening the formula?*" We encode this question as follows: First, we create a formula φ' that is a negation normal form (NNF) version of φ with every literal l_i replaced by $a_i \Rightarrow l_i$, where a_i is a fresh Boolean *assumption* variable associated with literal l_i. Note that if we set all a_i variables in φ' to $true$, we get a formula equivalent to φ. Now, we ask Z3 whether the formula

$$\varphi' \wedge \neg\varphi \wedge \bigwedge_i a_i$$

is unsatisfiable. This formula is unsatisfiable *iff* $\varphi \Rightarrow \varphi$ is valid, which is obviously the case. As a result, we can extract an UNSAT core over the assumption variables $\{a_i\}$; the UNSAT core indicates which literals in φ can be replaced by $true$ without changing its meaning. As Z3 is not guaranteed to return a minimal UNSAT core, we invoke it multiple times to get rid of as many literals as possible.

Second, after simplification, we examine the formula syntactically to eliminate as many variables as possible before passing the formula on to Z3. For instance, consider the formula

$$\forall x.\, x = y + z \vee \varphi,$$

where x does not appear in φ. In this case, we know that the literal $x = y + z$ can be replaced by *false*, thus eliminating the quantified variable x. We found that this simple heuristic eliminates a substantial number of variables, simplifying quantifier elimination for Z3.

Aggressive result postprocessing In addition to preprocessing the input to the quantifier elimination procedure, we also simplify the output of quantifier elimination. Given the complexity of quantifier elimination in LIA, the resulting formulas can be quite large and, in our experience, full of redundant subformulas.

Since our goal is to compute procedure specifications, we would like to produce simple specifications that are easy to examine by a human and/or efficient for reuse in verification. To achieve that, we applied the aforementioned UNSAT core simplification technique to get rid of redundant literals. For postprocessing, we go even further with simplification: we traverse the formula and attempt to remove redundant subformulas using local SMT checks.

Finding initial solutions The key component of our multi-abduction procedure is Cartesian decomposition. The maximal solution computed by Cartesian decomposition is determined by the initial solution (computed by INITSOLN). To compute *higher quality* specifications, we attempt to compute initial solutions that maximize the number of feasible paths. This ensures that computed specifications are not only maximal, but also permit as many program paths as possible. To do this, we implemented a greedy MAXSMT procedure that finds a model that maximizes the number of satisfiable disjuncts (program paths) in the formula χ in Algorithms 3 and 4.

Termination and non-linear multi-abduction Recall that in the presence of repeated occurrences of the same abducible (non-linear case), MULTIABDUCE may not terminate. For our set of benchmarks, the heuristic abduction loop, line 26 of Algorithm 4, always forced our algorithm to find a maximal solution. That is, we did not encounter non-terminating multi-abduction problems in practice.

7. Evaluation

Our evaluation is designed to assess (*i*) the efficiency and performance of our multi-abduction procedure and (*ii*) the quality of the synthesized specifications. For these purposes, we performed the following two sets of experiments:

E1 Our first experiment is primarily designed to evaluate and study the performance of our multi-abduction procedure. We apply the multi-abduction procedure to problems generated by (*i*) a parameterized set of microbenchmarks of growing size and (*ii*) a set of standard verification benchmarks that we use in a program repair scenario.

E2 Our second experiment evaluates the performance of our algorithm and the quality of synthesized specifications. For this, we used a collection of Windows device drivers from the software verification competition, SV-COMP [16]. For these programs, we eliminated kernel routine and initialization stubs, and synthesized maximal specifications that maintain correctness of the drivers. Stubs are typically manually written to abstract OS routines and avoid analysis of the whole OS kernel in the process of analyzing a single driver. Our evaluation demonstrates the potential of our approach at automatically synthesizing specifications of kernel routines from their usage context. One potential scenario would be that human auditors can then certify such specifications for reuse in verification of other drivers.

To the best of our knowledge, no existing technique can synthesize maximal safe specifications for the kinds of programs we address here. For our benchmarks, synthesizing maximal specifications requires multi-abduction and cannot be performed using simple abduction alone. Specifically, our benchmarks (*i*) call the same procedure multiple times and/or (*ii*) call different procedures. Section 8 provides a detailed comparison with related works.

7.1 E1: Evaluating multi-abduction

We now discuss our evaluation of multi-abduction. We first present our results on parameterized microbenchmarks and then on the `locks` benchmarks from SV-COMP.

7.1.1 Parameterized microbenchmarks

Benchmarks description We created four microbenchmarks that are parameterized by the number of calls to unknown procedures. All four benchmarks involve linear arithmetic operations and exhibit different scenarios for our tool. We evaluated our implementation on programs of increasing size. Our goal with this set of benchmarks is to stress our multi-abduction technique and experiment with how far it can go by increasing input sizes. We first describe our microbenchmarks and the synthesized specifications.

Consider the `accumulator` benchmark in the top part of Figure 3. In every iteration of the loop, the unknown procedure `fun` returns the values of `x` and `y`, which are then added to `z`. This benchmark is parameterized by the number of times, n, lines l_1 and l_2 are repeated within the loop. The loop executes arbitrarily many times (as denoted by the condition $*$). Our goal is to find a specification for `fun` that ensures that `z > 0` after the loop exit. For every instantiation of this benchmark, our tool computes the maximal specification $r_1 + r_2 \geqslant 0$, where r_1 and r_2 denote the first and second return values of `fun`.

Unlike the `accumulator` benchmark, `sum` and `sum-arg` require synthesizing specifications of increasing size as we increase the parameter n. For `sum`, n determines the number of `fun`'s return values; our tool computes the maximal specification

$$\sum_{i=1}^{n} i * r_i \geqslant 10,$$

where r_i denotes the i'th return value of `fun`. Hence, the inferred specification ensures that `y >= 10` throughout execution.

The benchmark `sum-arg` is trickier. Here, we call `fun` n times with n different arguments, from 1 to n, and we want to ensure that the sum of all return values is 0. One might be tempted to say that a maximal specification is $r = 0$, that is, `fun` always returns 0. This is too strong, as we only care about the return value of `fun` when its argument is between 1 and n. For each instantiation of this benchmark, our tool computes the maximal specification

$$\bigvee_{i=1}^{n} \mathit{arg} = i \;\Rightarrow\; r = 0,$$

where *arg* denotes the input argument of `fun` and r represents its return value. This specification states that `fun` returns 0 when its input is between 1 and n; otherwise, it can return any value.

Finally, the benchmark `pareto` is similar to `sum`, but sets the values of $\mathtt{x}_1, \ldots, \mathtt{x}_n$ by making calls to n different procedures. In this case, there are many maximal specifications for $\mathtt{fun}_1, \ldots, \mathtt{fun}_n$ that are incomparable as per our partial order (Definition 2). Our tool thus returns one possible maximal specification of the form

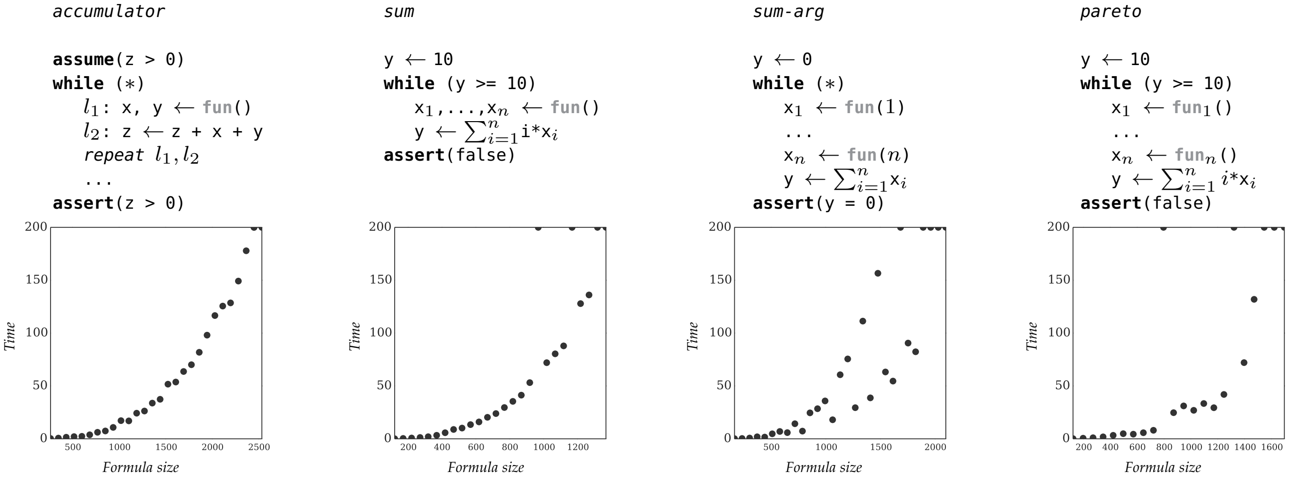
$$[\mathtt{fun}_1 \mapsto r_1 \geqslant c_1, \ldots, \mathtt{fun}_n \mapsto r_n \geqslant c_n],$$

where r_i is the return value of $\mathtt{fun}_i$, $c_i \in \mathbb{Z}$, and the following constraint is satisfied: $\sum_{i=1}^{n} i * c_i \geqslant 10$.

Results For each of the four benchmarks, we evaluated the performance of our multi-abduction procedure on the generated multi-abduction problem. (For these benchmarks, the multi-abduction solver is invoked once, as UFO always provides a sufficient DAG of counterexamples.) As we increase the size of each benchmark program, by increasing the parameter n, the size of the multi-abduction problem increases linearly. We measure formula size as the number of Boolean connectives in the DAG representation of the formula. Similarly, the numbers of variables and abducibles increase linearly with n.

The bottom part of Figure 3 represents the multi-abduction runtime results for each benchmark as the size of the formula increases. The time limit per benchmark is 3 minutes. Points on the top border are benchmarks that did not complete within time limit. We use two quantifier elimination strategies: (*i*) eliminating one variable at a time, by multiple calls to Z3, and (*ii*) variable partitioning—implemented internally in Z3 [19]. For each parameterized benchmark, we report results from the faster strategy.

Overall, our results indicate our multi-abduction technique's ability to scale to large formulas, with thousands of Boolean connectives, representing encodings of unrolled CFGs. For all of the formulas here, the number of quantified variables in the first call to ABDUCE (line 2 of Algorithm 4) is in the 100s. For instance, for formulas of size ~2000 from the `accumulator` benchmark, the number of abducibles is ~200, and the number of quantified variables is ~400 (out of a total of ~800 variables). In `sum-arg`, for formulas of size ~1500, we see ~200 abducibles and ~100 quantified variables (out of a total of ~500 variables). Notice the need for multi-abduction for these benchmarks: The same procedure `fun` is called multiple times, and there are calls to different procedures $\mathtt{fun}_i$. Thus, we require Cartesian and isomorphic decomposition to compute specifications of unknown procedures.



```
accumulator

assume(z > 0)
while (*)
    $l_1$: x, y ← fun()
    $l_2$: z ← z + x + y
    repeat $l_1, l_2$
    ...
assert(z > 0)
```

```
sum

y ← 10
while (y >= 10)
    $x_1, ..., x_n$ ← fun()
    y ← $\sum_{i=1}^{n}$ i*$x_i$
assert(false)
```

```
sum-arg

y ← 0
while (*)
    $x_1$ ← fun(1)
    ...
    $x_n$ ← fun($n$)
    y ← $\sum_{i=1}^{n} x_i$
assert(y = 0)
```

```
pareto

y ← 10
while (y >= 10)
    $x_1$ ← $fun_1$()
    ...
    $x_n$ ← $fun_n$()
    y ← $\sum_{i=1}^{n}$ $i$*$x_i$
assert(false)
```

Figure 3: Parameterized microbenchmarks and their results. Time is in seconds.

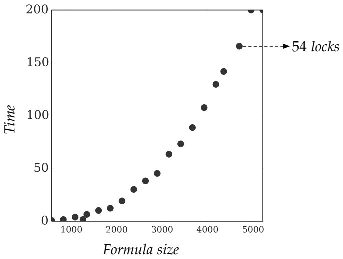


Figure 4: Results for locks benchmarks

7.1.2 Locks benchmarks

Benchmark description We also evaluated our approach on the locks benchmarks of SV-COMP (from the locks group), a set of benchmarks designed as challenge programs for automated software verifiers [18]. These benchmarks non-deterministically acquire and release a number of locks in a loop. The number of paths in the program increases exponentially in the number of locks. For each lock, we abstracted the code that checks whether the lock is acquired as a *unique* unknown procedure and used our tool to synthesize a specification under which the program is correct. Hence, the number of unknown procedures in these benchmarks is equal to the number of locks. The setup here mimics a *program repair* scenario: The assumption is that the code that checks whether the lock is acquired is "*suspect*" [37], and we are asking, "*What can we replace this code with in order to make the program correct?*"

Our tool synthesized a weaker specification for the abstracted code, instilling more non-determinism in these benchmarks while maintaining their correctness. Specifically, the checking code for each lock $\mathtt{lk}_i$ returns *true* if $\mathtt{lk}_i$ is not equal to 1, and *false* otherwise. Our tool returns the slightly weaker specification

$$\mathtt{lk}_i = 1 \Rightarrow \mathit{false},$$

which implies that the procedure can return *true* or *false* when $\mathtt{lk}_i \neq 1$. At first glance, this specification looks suspiciously weak, but a closer examination of the benchmark shows that $\mathtt{lk}_i$ is always not equal to 1 when the checking code is invoked. Thus, our tool synthesizes a specification that returns any value when $\mathtt{lk}_i$ is 1.

Results Figure 4 shows the amount of time (in seconds) multi-abduction takes as we increase the number of locks. The original benchmark suite contains programs with 5 to 15 locks; here, we extend the suite with new programs with up to 60 locks. As with the parameterized benchmarks, only one multi-abduction call is made. The x-axis shows the size of the formula as we increase the number of locks. Within the short time limit of 3 minutes, our multi-abduction solver is able to scale to problems with ∼5000 Boolean connectives, ∼160 abducibles, and ∼1500 quantified variables (generated from a program with 54 locks). Observe the need for multi-abduction here: We need to synthesize a specification for multiple procedures, one for each lock, and the same procedure might appear multiple times along control-flow paths emitted by the verifier. We would like to note that this set of benchmarks was specially crafted as a challenge to software verification tools: they require precise path reasoning; thus, a symbolic representation of paths is needed. Here, we do not only verify these benchmarks, we go one step further and synthesize maximal specifications for unknown code.

7.1.3 Summary

Our results demonstrate our multi-abduction procedure's ability to scale to large formulas with thousands of Boolean connectives and hundreds of variables. Our experience with quantifier elimination indicates the importance of the order in which variables are eliminated. Going forward, we would like to investigate variable elimination ordering strategies, with an emphasis on our specification synthesis domain. Our tool is able to produce formulas in SMT-LIB format, which we will release to drive future research in decision procedures of quantified formulas and quantifier elimination.

7.2 E2: Specification synthesis for device drivers

Description We now turn our attention to our second set of experiments, which target device drivers. In device driver verification, stubs for OS kernel routines are typically manually written to avoid analysis of complex kernel code. Indeed, this is the case for the Static Driver Verifier (SDV) [39], Microsoft's commercial product for driver developers, which uses the bounded verifier Corral [40].

In this section, we study the applicability of our proposed technique for automatically synthesizing specifications of kernel routines and initialization code for a standard suite of Windows drivers that are part of the software verification competition (SV-COMP). Here, we target the preprocessed benchmarks, from ntdrivers-simplified set, in which heap manipulation has been abstracted for the purposes of functional verification.

For evaluation, we identified all kernel routine stubs in these drivers: IofCallDriver, IofCompleteRequest, KeWaitForSingleObject, and KeSetEvent. These routines deal with I/O management and other core kernel issues, and are invoked by all drivers in

		Multi-abduction statistics		
`Driver/routine`	**Time**	**#Abds**	**DAG size**	**#Q–vars**
`kbfilter-s1`				
`init`	1	7	498	113–120
`IofCallDriver`	0.1	1	137	21–31
`IofComplRequest`	0.03*			
`KeSetEvent`	0.1	1	104	21–26
`KeWaitFSObject`	0.1	1	123	19–32
`kbfilter-s2`				
`init`	3	7	987	236–243
`IofCallDriver`	0.1	1	165	29–39
`IofComplRequest`	0.04*			
`KeSetEvent`	0.1	1	132	29–34
`KeWaitFSObject`	0.1	1	151	27–40
`floppy-s3`				
`init`	5	6	1757	382–388
`IofCallDriver`	16	9	1398	241–317
`IofComplRequest`	8	4	1579	330–343
`KeSetEvent`	1	3	1551	318–333
`KeWaitFSObject`	2	4	1693	316–368
`floppy-s4`				
`init`	34	6	2767	628–634
`IofCallDriver`	27	11	2282	436–529
`IofComplRequest`	33	9	2535	548–576
`KeSetEvent`	3	3	2502	546–561
`KeWaitFSObject`	4	4	2644	544–596
`diskperf`				
`init`	145	7	2075	432–439
`IofCallDriver`	15	9	1356	220–303
`IofComplRequest`	15	5	1548	313–329
`KeSetEvent`	1	6	1573	305–335
`KeWaitFSObject`	5	6	1876	322–397
`cdaudio`				
`init`	184	6	5251	1172–1178
`IofCallDriver`	53	11	2972	596–696
`IofComplRequest`	175	17	3723	817–869
`KeSetEvent`	5	3	3526	795–810
`KeWaitFSObject`	19	8	3842	788–892

Table 1: Results for device drivers. Time is in seconds.

the `ntdrivers` benchmarks.[2] Also, all drivers contain an initialization stub, `init`, invoked at the start of execution, to set initial values for global variables, flags, etc. For both the kernel routine stubs and the initialization stubs, we applied our technique to synthesize a maximal specification that ensures driver correctness, with respect to properties encoded in the drivers. That is, we removed the stubs and attempted to automatically synthesize their specifications from the routines' usage contexts, and we asked the following questions: (*i*) How efficient is the synthesis process? (*ii*) Do the synthesized specifications capture all behaviors of (i.e., over-approximate) the manually written stubs?

Synthesizing safe initialization specifications Synthesizing the initialization procedure of a device driver is an instance of input-filter synthesis: We infer the set of parameters for which the device driver behaves correctly. For each initialized global variable g in `init`, we replaced g's initialization code with $g \leftarrow \texttt{get}_g()$, where $\texttt{get}_g$ is an unknown procedure. Thus, by synthesizing a maximal specification for each $\texttt{get}_g$, we know the range of values that g must lie within in order to ensure correctness of the driver. Note that, for `init` specifications, we utilize multi-abduction to synthesize a specification per initialized variable. Simple abduction alone can only compute a monolithic formula over all uninitialized variables.

Table 1 shows the time it took to synthesize the specifications for all global variables for each driver in our set. The table also shows statistics for the final multi-abduction query: the number of abducibles (**#Abds**), the size of the problem (**DAG size**), and the number of quantified and total variables in the first call to ABDUCE (**#Q–vars**). For `init`, the number of abducibles (**#Abds**) corresponds to the number of initialized variables. For each driver, our synthesized specifications for procedures $\texttt{get}_g$ were one of the following: (*i*) a specific value for g, equal to the one in the initialization routine; (*ii*) a range of values for g that includes the one in the initialization routine; or (*iii*) the specification $true$, which indicates that g does not affect the correctness of the procedure. For all the drivers, our techniques was able to compute a maximal specification for 6-7 procedures in a small amount of time: 1 to 184 seconds.

Synthesizing kernel routine stubs In each driver, we replaced each kernel routine stub with an unknown procedure whose arguments are the arguments of the kernel routine and the global variables read by the routine; the return values of the unknown procedure are its actual return value and values for all global variables it updates. Using our approach, we synthesized maximal safe specifications for these procedures.

Table 1 shows the amount of time it took to synthesize a specification for each kernel routine. (Routine `IofCompleteRequest` is not required for correctness of drivers `kbfiltr-s*`, i.e., its maximal specification is $true$; this is detected by our front-end.) Consider the driver `cdaudio`; our technique computes a specification for `KeWaitForSingleObject` within 15 seconds, having to solve the (final) multi-abduction query containing ~3000 Boolean operators and ~800 quantified variables. We manually inspected synthesized specifications and in all cases found that they are equivalent to the original stubs or allow strictly more behavior. For instance, for the routine `IofCallDriver` in `kbfiltr-s1`, our technique synthesized the disjunctive specification that states that `IofCallDriver` either updates the global variable `s` to the value of global variable `NP`, or returns the signal 259 and updates the value of `s` to that of `MPR3`. The specification does not mention the global variable `lowerDriverReturn`, which is updated in the given stub, meaning that its value is not needed for the correctness of the driver.

Summary Our evaluation indicates our technique's ability to synthesize maximal safe specifications of unknown procedures in realistic programs. We have demonstrated this on a standard set of Windows driver benchmarks, where, within a short amount of time, we synthesized specifications of kernel routines that over-approximate existing stubs. We believe that our results demonstrate the promise of our approach, and provide strong evidence that it is indeed possible to synthesize specifications of unknown procedures.

8. Related Work

In this section, we place our specification synthesis technique in the context of related work.

Specification inference There is a large body of work on learning procedure specifications [7, 9, 12, 13, 20, 27, 35, 41, 45, 48, 50, 53, 54]. Many of these techniques [7, 9, 13, 45, 48, 50, 53] use either client or library code to learn API-usage rules of libraries, which can then be used for verifying other clients of the same library. Here, we addressed an orthogonal problem: synthesizing logical specifications of unknown procedures that ensure client safety.

The works of Bastani et al. [12] and Zhu et al. [54] have a similar goal to ours, but in the context of taint-flow analysis. Bastani et al. present a CFL-based solution for computing taint-flow and aliasing specifications. Similarly, Zhu et al. use simple abduction to find the smallest number of assumptions on unknown procedures. In contrast to these works, our technique differs in two ways. First, we present a general logic-based technique that is not tied to taint-flow analysis or CFL-based analyses. Second, our technique ensures

[2] See the Windows Driver Kit (WDK) for the detailed API [3].

a universally maximal solution, whereas maximality of prior work is relative to an initial taint-flow analysis, which is likely imprecise.

Seghir and Kroening [49] and Cimatti et al. [23] compute a weakest safe precondition of a program or transition system. This problem can be modelled in our framework by inserting a single unknown procedure call at the beginning of the program, where the procedure call returns initial values for program parameters.

Work on *angelic verification* [20, 27] aims to find *reasonable* environment assumptions in order to suppress "*stupid*" false alarms. In the recent paper by Das et al. [27], the user is assumed to specify a set of acceptable specifications; the job of the verifier is to find a specification within the supplied set that suppresses bugs while being permissive (limits dead code). Our work is similar in that we look for specifications of unknown code that make the program correct. However, our work differs in that we present a general logical treatment of the problem. In particular, our multi-abduction algorithm enables automatic computation of maximal specifications, but *without user intervention*.

Abduction in analysis and verification Abduction made its way into computer science through artificial intelligence, where it was used for knowledge representation and reasoning [29, 43]. Here, we focus on use of abduction in program analysis and verification.

In the context of separation logic for shape analysis, *bi-abduction* was introduced in the seminal work of Calcagno et al. [22] as a means for inferring procedure preconditions that ensure memory-safe execution. Our work differs in a number of ways: First, we target first-order theories that admit quantifier elimination. Second, multi-abduction finds specifications of unknown procedures in addition to preconditions, whereas bi-abduction assumes that it has access to callee specifications. We believe that our approach can be profitably combined with bi-abduction in a bottom-up compositional reasoning fashion. For instance, the recently open-sourced tool, Infer [1], a commercial version of Calcagno et al.'s technique, does not perform precise reasoning over arithmetic operations, and may return false positives in such cases. By incorporating a first-order abduction mechanism alongside shape abduction, we can create a more precise analysis technique.

The work of Qin et al. [47] presents a combined shape-data abstract domain that can be used for inferring specifications of unknown procedures using a form of abduction over the abstract domain. In comparison, we (*i*) compute maximal specifications of unknown procedures, whereas Qin et al.'s work heuristically abduces a specification; and (*ii*) our abduction queries always find a solution if one exists, as we are not restricted by an abstract domain that loses information during forward propagation.

In program analysis, abduction appeared first in the work of Giacobazzi [31] in the context of logic programming. Giacobazzi defined a top-down abstract interpretation of logic programs that abduces properties of modules that ensure correctness of their composition. Our work is analogous, in the sense that we also ask, "*How should callees behave in order to ensure correctness?*" In comparison to Giacobazzi's work, we target imperative programs and do not constrain our analysis to an abstract domain; instead, we encode program semantics as first-order formulas and utilize multi-abduction to synthesize weakest possible procedure specifications necessary for ensuring correctness.

Horn-clause solving Our work shares similarities with works on solving *Horn clauses* [15, 32, 33]. Our technique can be viewed as a solver for *recursive* Horn clauses with *unconstrained* predicates: those that appear exclusively in the bodies of clauses, and not as heads. Intuitively, unconstrained predicates represent specifications of unknown procedures—computed by multi-abduction—and constrained predicates represent inductive invariants proving that the program is safe—computed by the verifier. Existing work on solving Horn clauses does not address the question of optimality and would allow trivial solutions that set unconstrained predicates to $false$, as allowed by Horn-clause semantics. In contrast, our multi-abduction solver ensures that we compute maximal, non-trivial solutions. Picking up a recursive Horn-clause solver—like HSF [32] or GPDR [36]—and applying it in our setting does not guarantee a useful solution. Indeed, in our experience, unconstrained predicates in GPDR are always set to *false*. Thus, our contribution in the context of Horn-clause solving is a recursive Horn-clause solver that ensures non-trivial, maximal solutions to unconstrained predicates.

Beyene et al. [14] present a game solving technique that is formalized as Horn-clause solving. The authors demonstrate their approach for program repair and synthesis of winning strategies in two-player games. Our approaches are closely related. For instance, in Beyene et al., program repair is formalized as finding a strengthening of a program's transition relation. In our setting, this is equivalent to finding a specification of an unknown procedure that restricts program executions to safe ones. In comparison with Beyene et al., our work does not require user-supplied linear-arithmetic templates for specifications of unknown procedures; instead, it synthesizes a maximal specification over a first-order theory, a much more challenging problem. On the other hand, Beyene et al. also handle temporal specifications, allowing them to stipulate that a synthesized procedure specification ensures total correctness.

Formula decomposition Veanes et al. introduced *monadic decomposition* [52], a technique that takes a formula $F(x, y)$ and computes k pairs of formulas $(A_i(x), B_i(y))$ such that $\bigvee_i(A_i(x) \wedge B_i(y)) \equiv F(x, y)$. Our formulation of Cartesian decomposition is similar to 1-monadic decomposition, but it is more general: We relax the equivalence constraint to $A(x) \wedge B(y) \Rightarrow \varphi$ and look for maximal solutions of A and B. Also, Cartesian decomposition does not restrict the decomposition to a pair of formulas, but allows an arbitrary number of unknowns. In addition, we consider isomorphic decomposition. It is also important to note that our Cartesian decomposition procedure computes a 1-monadic decomposition when it exists.

9. Conclusion

We studied the problem of synthesizing maximal safe specifications of unknown procedures and presented a logic-based solution that leverages multi-abduction. Our empirical evaluation demonstrates the effectiveness of this technique at finding useful procedure specifications for realistic programs.

We believe there are many exciting and important applications of our work; in the future, we plan to investigate the following:

- ***Bottom-up compositional analysis for shape-data*:** As discussed in Section 8, our abduction algorithms can be combined with the bi-abduction-based shape analysis of Calcagno et al. [22] to produce a combined shape-data analysis. In the future, we plan to study such a combination with the goal of reducing false positives in shape analysis.
- ***Compositional top-down verification*:** We also plan to investigate the effectiveness of our technique for top-down verification. We can use our technique to verify callers independently of callees and then separately check whether the callees satisfy the inferred maximal specifications. In comparison with forward abstract-interpretation-based, bottom-up, and interpolation-based analyses, this direction has received very little attention [31, 47].

Acknowledgements We would like to thank Kenneth McMillan for one of the examples, and Zachary Kincaid, Thomas Reps, and Ben Liblit for comments on an early draft of the paper. The second author is funded by NSF Award #1453386 and FA 8750-14-2-0270.

References

[1] Infer. `http://fbinfer.com/`.

[2] The LLVM compiler infrastructure. `http://llvm.org`.

[3] Windows driver kit (WDK). `https://msdn.microsoft.com/en-us/library/windows/hardware/ff557573(v=vs.85).aspx`.

[4] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig interpretation. In *SAS*, 2012.

[5] A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-approximations to Over-approximations and Back. In *TACAS*, 2012.

[6] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction-and interpolation-based software verification. In *CAV*, 2012.

[7] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, 2005.

[8] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[9] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, 2002.

[10] T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV*, 2001.

[11] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, 2005.

[12] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *POPL*, 2015.

[13] N. E. Beckman and A. V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*, 2011.

[14] T. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, 2014.

[15] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV*, 2013.

[16] D. Beyer. Status report on software verification - (Competition summary SV-COMP 2014). In *TACAS*, 2014.

[17] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *STTT*, (5-6), 2007.

[18] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software Model Checking via Large-Block Encoding. In *FMCAD*, 2009.

[19] N. Bjørner. Linear quantifier elimination as an abstract decision procedure. In *IJCAR*, 2010.

[20] S. Blackshear and S. K. Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *PLDI*, 2013.

[21] D. Brumley, H. Wang, S. Jha, and D. X. Song. Creating vulnerability signatures using weakest preconditions. In *CSF*, 2007.

[22] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *POPL*, (1), 2009.

[23] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Parameter synthesis with IC3. In *FMCAD*, 2013.

[24] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, pages 168–176, 2004.

[25] D. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, (91-99), 1972.

[26] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *SOSP*, 2007.

[27] A. Das, S. K. Lahiri, A. Lal, , and Y. Li. Angelic verification: Precise verification modulo unknowns. In *CAV*, 2015.

[28] L. de Moura and N. Bjrner. Z3: An efficient SMT solver. In *TACAS*. 2008.

[29] M. Denecker and A. C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, 2002.

[30] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. *PLDI*, 2012.

[31] R. Giacobazzi. Abductive analysis of modular logic programs. In *ISLP*, 1994.

[32] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF (C): a software verifier based on Horn clauses. In *TACAS*. 2012.

[33] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[34] S. Gulwani. Synthesis from examples. *WAMBSE*, (2), 2012.

[35] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/SIGSOFT FSE*, 2005.

[36] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.

[37] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, 2005.

[38] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, 2010.

[39] A. Lal. personal communication, 2015.

[40] A. Lal, S. Qadeer, and S. Lahiri. Corral: A solver for reachability modulo theories. In *CAV*, 2012.

[41] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *PLDI*, 2009.

[42] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard. Sound input filter generation for integer overflow errors. 2014.

[43] S. McIlraith. Logic-based abductive inference. Technical Report KSL-98-19, Knowledge Systems Laboratory, July 1998.

[44] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[45] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, 2002.

[46] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *CACM*, 1992.

[47] S. Qin, C. Luo, G. He, F. Craciun, and W. Chin. Verifying heap-manipulating programs with unknown procedure calls. In *ICFEM*, 2010.

[48] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. PLDI '07, 2007.

[49] M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *ESOP*, 2013.

[50] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. ISSTA, 2007.

[51] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[52] M. Veanes, N. Bjørner, L. Nachmanson, and S. Bereg. Monadic decomposition. In *CAV*, 2014.

[53] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. ICSE, 2006.

[54] H. Zhu, T. Dillig, and I. Dillig. Automated inference of library specifications for source-sink property verification. In *APLAS*, 2013.