Compositional Safety Refutation Techniques

Kumar Madhukar[1,3](✉), Peter Schrammel[2], and Mandayam Srivas[3]

[1] TCS Research, Pune, India
kumar.madhukar@tcs.com
[2] School of Engineering and Informatics, University of Sussex, Brighton, UK
[3] Chennai Mathematical Institute, Chennai, India

Abstract. One of the most successful techniques for refuting safety properties is to find counterexamples by bounded model checking. However, for large programs, bounded model checking instances often exceed the limits of resources available. Generating such counterexamples in a modular way could speed up refutation, but it is challenging because of the inherently non-compositional nature of these counterexamples. We start from the monolithic safety verification problem and present a step-by-step derivation of the compositional safety refutation problem. We give three algorithms that solve this problem, discuss their properties with respect to efficiency and completeness, and evaluate them experimentally.

1 Introduction

Divide-and-conquer approaches are considered to be the blue print solution to scale algorithms to large problems. Compositionality of proofs is the enabler of a map-reduce approach to verification. Compositional verification approaches based on contracts and summaries have been shown to tremendously increase scalability and productivity in real-world formal verification [2,12,19,27].

But what about refutation? Unlike verification, refutation algorithms are usually based on finding a violating execution trace, which seems to be inherently non-compositional. Consequently, the study of the compositional refutation problem is an under-explored area of research. Yet, solutions to this problem have significant impact on other research problems. As a motivation, we give here two algorithmic approaches in verification and testing that will be enabled by efficient compositional refutation algorithms:

– Property-guided abstraction refinement algorithms like CEGAR [6] need to decide whether counterexamples that are found in the abstraction are spurious or true counterexamples. The lack of compositional refutation techniques forces these algorithms to operate in a monolithic manner and is therefore an obstacle to scaling them to large programs.
– Automated test generation techniques based on Bounded Model Checking are successfully used in various industries to generate unit tests (e.g. [25]). However, they do not sufficiently scale to accomplish the task of generating

D. D'Souza and K. Narayan Kumar (Eds.): ATVA 2017, LNCS 10482, pp. 164–183, 2017.
DOI: 10.1007/978-3-319-68167-2_12

integration tests. Compositional refutation techniques achieve exactly this goal: they efficiently produce refutations (from which test vectors can be derived) on unit (module) level and enable their composition in order to obtain system level refutations, i.e. integration tests.

This paper is a first step in this direction and lays the base for a more systematic study of the problem domain.

Contributions. We summarise the contributions of the paper as follows:

- In order to place the problem in a wider context, we give an informal overview on how completeness relates to problem decomposition in safety refutation and verification (Sect. 3).
- We formalise the safety refutation problem in *horizontal decompositions*, e.g. procedure-modular decompositions, and characterise the compositional completeness guarantees of various algorithmic approaches (Sect. 4).
- We describe three refutation approaches with different degrees of completeness (Sect. 5) and give experimental results on C benchmarks, comparing their completeness and efficiency (Sect. 6).

2 Preliminaries

Program Model and Notation. We assume that programs are given in terms of acyclic[1] call graphs, where individual procedures f are given in terms of *deterministic*, symbolic input/output transition systems. F is the set of all procedures in the program. Since the handling of loops is orthogonal to the compositional aspect, we consider only loop-free procedures (respectively bounded unwindings of loops) in this paper. Thus, we simply denote the input/output relation of a procedure f as $T_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$. Inputs $\boldsymbol{x}^{in}$ are procedure parameters, global variables, and memory objects that are read by f. Outputs $\boldsymbol{x}^{out}$ are return values, and potential side effects such as global variables and memory objects written by f. Boolean guard variables (g) are used to model the control flow. Non-deterministic choices are encoded by additional input variables.

The relations T_f are given *as first-order logic formulae* over bitvectors and arrays, resulting from the logical encoding of the program semantics. Figure 1 gives an example of the encoding of a program into such formulae using the loop-free notation. The inputs $\boldsymbol{x}^{in}$ of *foo* are (y, g_6) and the outputs $\boldsymbol{x}^{out}$ consist of (r, g_7) where r is the return value. In addition to the inputs and outputs we need boolean guard variables g^{in}, g^{out} (here g_6, g_7) that are true if the entry and, respectively, exit of the procedure can be reached. They are handled like input/output parameters and have their actual counterparts in the guard variables in the caller (here, e.g. g_1, g_2 for the call foo_0 in *main*). Note that we consider exit in a procedure is not reachable, i.e., $\neg g^{out}$, if either the program is non-terminating or an assertion in a procedure is violated. Hence, the exit guard

[1] We consider non-recursive programs with multiple procedures (cf. model in [5]).

```
void main(int x) {
  if(x < 0) {
    x = foo(x);
    x = foo(x);
    bar(x);
  }
}

int foo(int y) {
  return y+1;
}
void bar(int z) {
  assert(z > 10);
}
```

$$\begin{aligned}
T_{main}((x_0,g_0),(g_5)) &\equiv g_1{=}(g_0 \wedge (x_0{<}0))\wedge\\
&\quad\; foo_0((x_0,g_1),(x_1,g_2))\wedge\\
&\quad\; foo_1((x_1,g_2),(x_2,g_3))\wedge\\
&\quad\; bar((x_2,g_3),(g_4))\wedge\\
&\quad\; g_5{=}(g_0 \wedge \neg(x_0{<}0) \vee g_4)\\
\mathit{Props}_{main} &\equiv \mathit{true}\\
T_{foo}((y,g_6),(r,g_7)) &\equiv (r{=}y{+}1) \wedge (g_6{=}g_7)\\
\mathit{Props}_{foo} &\equiv \mathit{true}\\
T_{bar}((z,g_8),(g_9)) &\equiv g_9{=}(g_8 \wedge (z{>}10))\\
\mathit{Props}_{bar} &\equiv g_8 \Rightarrow (z{>}10)
\end{aligned}$$

Fig. 1. Example program and its encoding

condition in the definition of a transition function includes assertion checks as in T_{bar}. We use a single static assignment (SSA) encoding, which gives a fresh name to each update of a variable if it is modified multiple times, such as for example in *main*.

Each call to a procedure h at call site i in a procedure f is modeled by a *placeholder predicate* $h_i(\boldsymbol{x}_i^{p\_in}, \boldsymbol{x}_i^{p\_out})$ occurring in the formula T_f for f. The placeholder predicate ranges over intermediate variables representing its actual input and output parameters $\boldsymbol{x}_i^{p\_in}$ and $\boldsymbol{x}_i^{p\_out}$, respectively. Placeholder predicates evaluate to *true* in the beginning, which corresponds to havocking the program variables in procedure calls. As the analysis progresses, they get strengthened by summaries. We later explain how we use the guard variables in performing this propagation. In procedure *main* in Fig. 1, the placeholder for the first procedure call to *foo* is $foo_0((x_0, g_1), (x_1, g_2))$ with the actual input and output parameters x_0, x_1, respectively, and the corresponding guard variables that encode whether the entry and exit of foo_0 are reachable. Let Props_f denote the property (assertion) in procedure f (e.g. the assertion in *bar* in Fig. 1). Note that we view these formulae as predicates, e.g. $T(\boldsymbol{x}, \boldsymbol{x}')$, with given parameters $\boldsymbol{x}, \boldsymbol{x}'$, and mean the $T[\boldsymbol{a}/\boldsymbol{x}, \boldsymbol{b}/\boldsymbol{x}']$ when we write $T(\boldsymbol{a}, \boldsymbol{b})$. Moreover, we write $\boldsymbol{x}$ and x with the understanding that the former is a vector, whereas the latter is a scalar.

CS_f is the set of call sites in procedure f, and the set of all call sites CS is $\bigcup_{f\in F} CS_f$. $fn(i)$ is the procedure called at call site i. We write X_f for the variables in T_f, and $\hat{X}$ for the entirety of variables in $T_{fn(i)}(\boldsymbol{x}_i^{in}, \boldsymbol{x}_i^{out})$ for all $i \in CS$.

Summaries, and Calling Contexts. Inter-procedural compositional proofs of a sequential program usually use a set of auxiliary predicates to define abstractions of loops and procedures. These abstractions are usually formally defined by means of a set of predicates – *invariants*, a *summary* and a *calling context* ($\mathit{CallCtx}_i$) for every procedure invocation h_i at call site i in a call-graph of the program. These predicates have the following roles: Invariants abstract the behaviour of loops inside functions. Summaries abstract the behaviour of called procedures; they are used to strengthen the placeholder predicates.

Calling contexts abstract the caller's behaviour w.r.t. the procedure being called. When analysing the callee, the calling contexts are used to constrain its inputs and outputs. The set of sub-traces corresponding to a function at a call site is characterised by a conjunction of the calling context and summary predicates associated with the function at that call site. We provide formal definitions for summaries and calling contexts below (invariants are not needed in this paper).

Definition 1. *For a procedure given by T_f we define:*

– *A* summary *is a predicate Sum_f such that:*

$$\forall X_f : T_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \Longrightarrow Sum_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$$

– *The* calling context *for a procedure call h at call site i in the given procedure is a predicate $CallCtx_i$ such that*

$$\forall X_f : T_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \Longrightarrow CallCtx_i(\boldsymbol{x}_i^{p\text{-}in}, \boldsymbol{x}_i^{p\text{-}out})$$

For instance, a summary for procedure *foo* in Fig. 1, is $Sum_{foo}((y, g_6), (r, g_7)) = (y{<}MAX \Rightarrow r{>}y)$.[2] A (forward) calling context for the first call to procedure *foo* in *main* is $CallCtx_{foo_0}((x_0, g_1), (x_1, g_2)) = (g_1 \Rightarrow x_0{<}0)$.

We observe that the guard variables are also used in defining summaries and calling contexts. They have the same meaning as in transition functions. The reason we have defined *CallCtx* over both input and output parameters is so we can propagate it in forward or backward directions.

3 Compositional Verification and Refutation Overview

A decomposition of a verification problem intuitively splits the original problem into a set of sub-problems that cover the original problem. The decomposition operator for the problem has a corresponding composition operator for composing the results obtained from the sub-problems in order to obtain a solution of the original problem.

In terms of program executions, a decomposition can be viewed as a way a proof of verification splits the behaviour, i.e. the set of all execution traces of a program, in constructing the proof. Consider a safe version of the code in Fig. 1 where the assertion in *bar* is changed to $z \leq 10$. A safety proof for the program can be constructed hierarchically by using the following summaries for *foo* and *bar*: $Sum_{foo}((y, g_6), (r, g_7)) = (r{=}y+1 \wedge g_6{=}g_7)$ and $Sum_{bar}((z, g_8), (g_9)) = (g_9 \Rightarrow z \leq 10)$. Then, the proof for *main* can be constructed using the recursive Algorithm 1. The proof for the leaves (*foo* and *bar*) involves showing their transition functions imply their respective summary. Proof composition for a non-leaf procedure will use the caller summaries to similarly construct a proof (a summary) for the caller. In our example, the program is indeed proved safe as the algorithm constructs a Sum_{main}, which, in this case, can be a suitable

[2] MAX denotes the maximum possible value in the type of y.

Algorithm 1. Composition operator for summaries

```
1: procedure COMPOSE($f$)
2:     for all $i \in CS_f$ do                         ▷ $CS_f$ are the call sites in procedure $f$
3:         $Sum_{fn(i)} \leftarrow$ COMPOSE($fn(i)$)       ▷ $fn(i)$ is the procedure called at call site $i$
4:     $Sum_f \leftarrow proof(f)$       ▷ uses $Sum_{fn(i)}$, $i \in CS_f$ and proof composer operator
5:     return $Sum_f$                                   ▷ $Sum_f$ can be cached
```

abstraction of the transition function for *main*, that is not *false*, while checking that the constructed summaries verify all the embedded properties.

For sequential programs, decompositions can be *vertical* or *horizontal*. A vertical decomposition usually focuses on entire execution traces and splits the behaviour of the program into subsets of end-to-end traces, e.g. program slicing [16]. A *horizontal* decomposition is usually based on a syntactic decomposition of the program e.g. into procedures. This paper focuses on solving the refutation problem with horizontal decompositions.

The challenge in automating horizontal compositional verification lies in synthesising a set of precise summary predicates for the procedures in the call graph. Note that in the program in Fig. 1, it was essential to constrain the input z to *bar* to be $(z \leq 1)$ to get a proof. This effort is made harder if the code has loops, which require *invariants* and use of abstractions. The calling contexts and summaries can be mutually dependent even for non-recursive programs. In general, one requires iterative fix-point computation on the call-graph structure, possibly using abstraction and refinement. A pre-requisite for performing abstraction refinement is the ability to refute safety and check for spurious counterexamples also in a modular and efficient fashion, which is the goal of this paper.

A Practical View of the Modular Refutation Problem. Consider the example in Fig. 1 in Sect. 2. This program is unsafe because when *bar* is called the actual argument to it that takes the place of z can at most be only 1. The question is if we can arrive at this refutation modularly. Analysing procedure *bar* in isolation indeed gives a counterexample, which could be possibly spurious.

Instantiated on the example in Fig. 1, a refutation involves checking $\neg\forall z, g_8 : g_8 \Rightarrow (z > 10)$. A counterexample could be $g_8 \wedge z = 5$, for example. The question is now how to decide whether this counterexample is spurious or not, and to find a valid counterexample if one exists. For instance, $z = 5$ turns out to be spurious if we consider the whole program because it clashes with $x_0 < 0$ in *main*. However, $z = -8$ would be a valid counterexample.

The set of *local* counterexamples found in a procedure f might contain many counterexamples that are spurious for the whole program, i.e. they are infeasible from the entry point of the program. A definite answer to this question cannot be given by looking at the local problems alone, but only by analysing the global one. This is the reason why refutation in horizontal decompositions is hard — unlike refutation in vertical decompositions where a refutation of the local problem implies the refutation of the global one.

Intuitively, the negation of the *assertion* has to be hoisted up along the error path to the entry point of the program. If the obtained weakest precondition for the violation of the assertion is not *false*, then the counterexample is feasible. Propagating up the *counterexample* itself is not sufficient to decide spuriousness as illustrated above.

4 Formalising Horizontal Compositional Refutation

In this section we formalise the problem of safety refutation for sequential programs. To simplify the presentation we focus on *loop-free* programs. The formalisation for programs with loops is structurally similar, but in addition, requires the handling of invariants, which is orthogonal to the compositional aspect.

We give three different formalisations – the first corresponds to a monolithic approach, and the remaining two correspond to compositional approaches.

4.1 Monolithic Safety Refutation Problem

For non-recursive programs, since one can always inline every procedure call at its call site, we can replace every call by recursively inlining its body. Then, to *refute* safety we have to show unsatisfiability of the following formula:

$$\forall \hat{X} : \bigwedge_{j \in CS} g^{in}_{f_{entry}} \wedge T_{fn(j)}(\boldsymbol{x}^{in}_j, \boldsymbol{x}^{out}_j) \wedge \mathit{InlineSums}_{fn(j)} \Rightarrow \mathit{Props}_{fn(j)}(\boldsymbol{x}_j) \quad (1)$$

where

- $\mathit{InlineSums}_f$ is $\bigwedge_{i \in CS_f} \mathit{InlineSum}_{fn(i)}(\boldsymbol{x}^{p\_in}_i, \boldsymbol{x}^{p\_out}_i)$,
- $\mathit{InlineSum}_f(\boldsymbol{x}^{in}_f, \boldsymbol{x}^{out}_f)$ is $T_f(\boldsymbol{x}^{in}_f, \boldsymbol{x}^{out}_f) \wedge \mathit{InlineSums}_f$,
- $\hat{X}$ is the entirety of variables in (1),
- and the conjunction with $g^{in}_{f_{entry}}$ states that the entry procedure is reachable.[3]

Alternatively, we can write:

$$\begin{aligned} &\exists \overbrace{\mathit{Sum}_f, \ldots :}^{\text{for all } f \in F} \bigwedge\nolimits_{f \in F} \forall X_f : \\ &\quad \big(g^{in}_{f_{entry}} \wedge T_f(\boldsymbol{x}^{in}_f, \boldsymbol{x}^{out}_f) \wedge \mathit{Sums}_f \Longrightarrow \mathit{Props}_f(\boldsymbol{x}_f)\big) \\ &\quad \wedge \big(T_f(\boldsymbol{x}^{in}_f, \boldsymbol{x}^{out}_f) \wedge \mathit{Sums}_f \Longleftrightarrow \mathit{Sum}_f(\boldsymbol{x}^{in}_f, \boldsymbol{x}^{out}_f)\big) \end{aligned} \quad (2)$$

where Sums_f is $\bigwedge_{i \in CS_f} \mathit{Sum}_{fn(i)}(\boldsymbol{x}^{p\_in}_i, \boldsymbol{x}^{p\_out}_i)$.

This formulation uses a predicate Sum_f to exactly express the behaviours of each procedure f. (1) and (2) are equisatisfiable, i.e., 1 is satisfiable iff 2 is satisfiable. The existential quantifier in (2) can be uniquely eliminated by recursively replacing the Sum_f predicates by left-hand side of the equivalence in the last line in (2), obtaining (1). Note that solving (1) is NP-complete, whereas

[3] This amounts to using $T_{f_{entry}}[\mathit{true}/g^{in}_{f_{entry}}]$ as the transition relation of f_{entry}.

solving (2) is PSPACE-complete. However, (1) may be exponentially larger (in the number of variables) than (2).

Both versions are *monolithic* because they consider the entire program as a whole. In particular, (2) finds summaries *globally*, i.e. for the whole program.

Also note that, proving unsatisfiability of (2) shows the inexistence of a verification proof, but it does not directly allow us to derive a counterexample in terms of an execution trace because of the universal quantification of the variables. Moreover, showing unsatisfiability of (2) is difficult because it involves proving the inexistence of summary predicates. For this reason, many practical techniques, such as SAT-based Bounded Model Checking use (1) (considering bounded unwindings for programs with loops in order to make them loop-free). Note that negating (1) results in an existentially quantified problem, whose satisfiability witnesses a refutation in the form of values for the variables $\hat{X}$.

However, solving (1) monolithically is often intractable. Therefore, we want to decompose the problem into smaller subformulae that are faster to solve. (2) is amenable to decomposition, but it does not allow us to approximate the summaries with the help of abstractions (because of $\Leftrightarrow$ in last line). Therefore we give a third formulation of the monolithic problem that additionally uses *calling contexts*. The calling context for the entry procedure is $g^{in}_{f_{entry}}$.

$$\begin{array}{l}\exists \overbrace{Sum_f, CallCtx_f, \ldots :}^{\text{for all } f\in F} \bigwedge_{f\in F} \forall X_f : \\ \big(CallCtx_f(\boldsymbol{x}^{in}_f, \boldsymbol{x}^{out}_f)\wedge \\ T_f(\boldsymbol{x}^{in}_f, \boldsymbol{x}^{out}_f) \wedge Sums_f \implies Props_f(\boldsymbol{x}_f)\wedge \\ \qquad\qquad\qquad\qquad\qquad\quad Sum_f(\boldsymbol{x}^{in}_f, \boldsymbol{x}^{out}_f)\wedge \\ \qquad\qquad\qquad\qquad\qquad\quad \bigwedge_{j\in CS_f} CallCtx_{fn(j)}(\boldsymbol{x}^{p\_in}_j, \boldsymbol{x}^{p\_out}_j)\big)\end{array} \tag{3}$$

Equation (3) is also equisatisfiable with (2), although (3) admits more solutions to Sum_f including those that are over-approximations adequate to prove the properties. To see this, if (2) is satisfiable, the precise solution of (2) for Sum_f can be used to satisfy (3) by plugging it in for both $CallCtx_f$ and Sum_f in (3). If (2) is unsatisfiable, then so is (3) because one or all behaviour included in Sum_f solution of (2) violates one of the properties. Then, every solution to (3) would violate the properties as they are over-approximations of the precise summaries.

4.2 Modular Safety Refutation Problem

Let us now have a look at the *horizontal* decomposition following the procedural structure of the program. The goal is to compute the summaries Sum_f for each f while considering only f and the summaries for the procedures called in f. We can attempt at achieving this by flipping the existential quantifier ($\exists Sum_f$) and the top-level conjunction ($\bigwedge_{f\in F}$ in (3)). However, this does not result in an equisatisfiable formula because existential quantification does not distribute over conjunctions. Therefore, we need an alternative formulation to solve the existential query per procedure. One approach is to search for a minimal solution for summaries and calling contexts occurring within each calling site of procedure f for a given context for f that satisfies all the embedded properties in f as shown in 4. I.e. for each $f \in F$ we have:

$$\begin{aligned}
&\min \mathit{Sum}_f, \overbrace{\mathit{CallCtx}_j, \dots}^{\text{for all } j \in CS_f} : \forall X_f : \\
&\big(\mathit{CallCtx}_f(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out}) \wedge \\
&\ T_f(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out}) \wedge \mathit{Sums}_f \implies \mathit{Props}_f(\boldsymbol{x}_f) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Sum}_f(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out}) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad \textstyle\bigwedge_{j \in CS_f} \mathit{CallCtx}_j(\boldsymbol{x}_j^{p\text{-}in}, \boldsymbol{x}_j^{p\text{-}out})\big)
\end{aligned} \tag{4}$$

$\min P : F(P)$ is defined w.r.t. implication order for a formula F involving predicates P, i.e. as $\exists P : F(P) \wedge \forall P' : (P' \Rightarrow P) \Rightarrow \neg F(P')$. Note that $\min P$ is not unique in a partial order. (4) gives a solution for Sum_f and the calling contexts for all embedded calling sites relative to a $\mathit{CallCtx}_f$, assuming there is a minimal solution for all embedded procedures. But, we have not broken the dependency between calling contexts and summaries. Solving this problem requires computing a fixed point in the composition operator (presented below) and computing minimal solutions for the summary and calling context predicates. That is, what has been an existential second-order satisfaction problem in (3), has now become a second-order minimisation ($\exists\forall$) problem. The reason for this is that the mere existence of a solution for Sum_f and $\mathit{CallCtx}_{fn(j)}$ does not prove that the overall verification problem holds. Therefore, we pessimistically have to assume that we require the *exact* calling contexts and summaries in order to *decide* the problem during proof composition.

The proposed proof composition operator (*compose*) with calling contexts is shown in Algorithm 2 and is more complex than Algorithm 1. The idea is to use the call graph of the program to compute the minimal calling context for each call site of procedure call of f piecewise in a top-down fashion use that calling context to compute a piecewise minimal summary for f for that call site (note the conjunction on Line 12 of Algorithm 2) consistent with all the properties in f. The piecewise summaries and contexts are combined disjunctively as they are built, which takes care of the dependency between summary and calling contexts. In the algorithm, each time *compose* is called recursively for f, it is called with a new piece of entry calling context for f and (4) is solved with summaries computed up to that point for the procedures in the body of f. Solving the equation smay result in new contexts for each call site (if any) inside f and a new piece of summary for f all of which are accumulated.

For a program with entry f_{entry}, a proof can be constructed by calling $\mathit{compose}(f_{entry}, g^{in}_{f_{entry}})$. The calling context $g^{in}_{f_{entry}}$ means that the entry procedure is reachable. The calling context of all embedded functions are initialised to *false* as that is the least element and also makes everything following the first call site unreachable. The summary for each f is initialised to $\neg g_f^{out}$, meaning that its exit is not reachable and hence execution cannot continue beyond any call to f. This initial value for summary has the effect of blocking analysis of all functions following f in the code until a piecewise summary is computed for f.

Algorithm 2. Composition operator with calling contexts

```
1: global $Sum_f \leftarrow \neg g_f^{out}$ for all $f \in F$
2:        $CallCtx_f \leftarrow false$ for all $f \in F$
3: procedure $compose(f,\ CallCtx_f^*)$
4:     while $true$ do                                          ▷ Repeat until fixed point reached
5:         Solve (4) for $f$ with $CallCtx_f^*$ as $CallCtx_f$   ▷ { obtain $Sum_f$ and $CallCtx_j$ for all $j \in CS_f$ }
6:         for all $j \in CS_f$ do                              ▷ join calling contexts for $fn(j)$
7:             $CallCtx_{fn(j)} \leftarrow CallCtx_{fn(j)} \vee CallCtx_j(\boldsymbol{x}_{fn(j)}^{in}, \boldsymbol{x}_{fn(j)}^{out})$
8:         if $CallCtx_{fn(j)}$ for all $j \in CS_f$ has not changed then
9:             return $Sum_f$
10:        for all $j \in CS_f$ for which $CallCtx_{fn(j)}$ has changed do
11:            $Sum_j \leftarrow compose(fn(j), CallCtx_j(\boldsymbol{x}_{fn(j)}^{in}, \boldsymbol{x}_{fn(j)}^{out}))$
12:            $Sum_{fn(j)} \leftarrow Sum_{fn(j)} \vee (CallCtx_j(\boldsymbol{x}_{fn(j)}^{in}, \boldsymbol{x}_{fn(j)}^{out}) \wedge Sum_j)$
13:                                                             ▷ join summaries for $fn(j)$
```

Observe that, as opposed to monolithic (3) where the fixed point computation for resolving the mutually dependent summary and calling context predicates (cf. [23]) is done within the solver for solving the monolithic formula, the fixed point in the modular version must be computed during the composition of the individual results. I.e. we have to saturate the Sum_f and $CallCtx_f$ predicates.

Theorem 1. *We obtain $Sum_{f_{entry}} = false$ using Algorithm 2 iff (3) is unsatisfiable. I.e. horizontal decomposition is sound and complete.*

Proof (sketch): We prove this by induction on the depth (k) of the top-level function in the call graph of the program.

For the base case ($k = 0$), there is only one procedure call - the call to the entry procedure, f_{entry}. Since the calling context of f_{entry} is $g_{f_{entry}}^{in}$, and there are no other procedure calls, it is evident that computing $Sum_{f_{entry}}$ from Algorithm 2 effectively reduces to finding it by solving (4) (line 5 of Algorithm 2), with *Sums* and $CallCtx_j$ not present. This makes Eqs. (4) and (3) identical and hence the theorem follows trivially.

Proceeding with the induction step for $k+1$ assuming the theorem holds for all functions in the call graph with depth $\leq k$. That is, we assume as hypothesis the summary computed by *compose* satisfies theorem for all function calls in the body of f for all contexts. Suppose (3) is unsatisfiable. We will argue that Algorithm 2 must return *false*.

If (3) is unsatisfiable then there must be at least one function (either the top-level function or something deeper in the call graph) that is unsatisfiable. Suppose it is one of the called functions, say h, that is unsatisfiable. Then, by our induction hypothesis, the algorithm will return *false* for Sum_h. The moment one of the embedded summaries becomes *false* our algorithm immediately saturates because Algorithm 2 is trivially satisfiable with minimal solution of *false* for

Sum_f. If (3) is unsatisfiable because the top-level function f is unsatisfiable, then it must be because $Props_f$ is inconsistent with T_f. In this case, Algorithm 2 can only return *false*.

4.3 Modular Safety Refutation with Witnesses

(4) suffers from the same problem as (3) that we cannot extract counterexamples in terms of an execution trace in case of a refutation because the formulae are unsatisfiable for refutations. Therefore we give next a formulation and a corresponding composition operator that produces refutation witnesses. The idea here is to compute piecewise contexts and summaries backwards starting from exit points of each procedure, much like a weakest-precondition computation works. Additionally, we start with negation of properties and compute maximal summary and contexts that possibly lead the program to an error state. In other words, a summary computed for f represent maximal symbolic witness to all the states reachable to safety violation. Such a summary can be obtained as maximal solutions to the equation shown in 5.

$$\begin{array}{l}
\max Sum_f, \overbrace{CallCtx_j, \ldots}^{\text{for all } j \in CS_f} : \forall X_f : \\
\quad Sum_f(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out}) \wedge \\
\quad \bigwedge_{j \in CS_f} CallCtx_j(\boldsymbol{x}_j^{p\text{-}in}, \boldsymbol{x}_j^{p\text{-}out}) \Longrightarrow (CallCtx_f(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out}) \vee \neg Props_f) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad T_f(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out}) \wedge Sums_f
\end{array} \tag{5}$$

where $\max P.F(P)$ is defined as usual: $\exists P.F(P) \wedge \forall P'.(P \Rightarrow P') \Rightarrow \neg F(P')$.

(5) describes maximal solutions for the summary and calling contexts that are *contained* in the behaviour of the procedure. That is the reason the predicates for the summary and the calling contexts (for the called functions) appear on the left-hand side of the implication and the transition relation is on the right-hand side, i.e. reversed in comparison with (4). The disjuncts in the first part of the consequent of (5) are the sources of safety violations: these are safety violations in the caller (which are propagated by $CallCtx_f$), and safety violations in f itself ($\neg Props_f$). Safety violations in callees are propagated through the summaries. Both these are constrained to be consistent with the transition relation of f (with current summaries plugged in for the called functions), which ensures spurious errors are not propagated upwards.

We use the composition operator as in Algorithm 2, but with the following modifications to the initialization. We call this composition operator *compose'* or Algorithm 2' from now on.

- Initially, $Sum_f \leftarrow \neg g_f^{in}$ for all $f \in F$, meaning that the entry of f is not backwards-reachable.
- In Line 5, we solve (5).

The calling contexts for all embedded functions are initialized to *false* as before except for the top-level function f_{entry}. A refutation is constructed by computing $\mathit{compose}'(f_{entry}, \neg g^{out}_{f_{entry}})$. The calling context $\neg g^{out}_{f_{entry}}$ of f_{entry} means that we cannot reach the regular exit of the entry procedure if there is a property violation. If there are no property violations at this level (or no properties), then this choice for top-level context would still work as the second conjunct in Eq. 5, which denotes the transition relation, would ensure the precise contexts propagated to the first embedded call site from exit point. The choice of initial summary of $\neg g^{in}_f$ for all embedded functions will ensure that the summaries are generated in order of dependency of function calls backward from the exit point.

Theorem 2. *We obtain $\mathit{Sum}_{f_{entry}}$ using Algorithm 2' such that $\exists \boldsymbol{x}^{in}, \boldsymbol{x}^{out} : g^{in}_{f_{entry}} \wedge \mathit{Sum}_{f_{entry}}(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$ iff (3) is unsatisfiable.*

Note that the conjunction with $g^{in}_{f_{entry}}$ projects the summary on the inputs, which must be satisfiable to have a refutation.

Proof (sketch): In contrast to Algorithm 2 with (4), Algorithm 2' uses (5) that computes the maximal solutions for the summary and calling contexts contained in the program behaviour. The summaries and calling contexts are computed such that their projection on the input variables of a procedure is the weakest precondition w.r.t. the properties (*Props*), whose complement is the refutation. Thus at the entry function, f_{entry}, we get $\mathit{Sum}_{f_{entry}}(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$ as weakest precondition for the negation of the property such that $g^{in}_{f_{entry}} \wedge \mathit{Sum}_{f_{entry}}(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$ is satisfiable iff (3) is unsatisfiable.

4.4 Worked Example

Let us consider the example in Fig. 1, but with the conditional in line 2 being $x < 10$. We start with $\mathit{Sum}_{main}((x_0, g_0), (g_5)) = \neg g_0$, $\mathit{Sum}_{foo}((y, g_6), (r, g_7)) = \neg g_6$, $\mathit{Sum}_{bar}((z, g_8), (g_9)) = \neg g_8$, and $\mathit{CallCtx}^*_{main}((x_0, g_0), (g_5)) = \neg g_5$, $\mathit{CallCtx}_{foo}((y, g_6), (r, g_7)) = \mathit{false}$, $\mathit{CallCtx}_{bar}((z, g_8), (g_9)) = \mathit{false}$.

The composition operator is called for *main*. We solve (5):

$$\begin{array}{ll}
\multicolumn{2}{l}{\max \mathit{Sum}_{main}, \mathit{CallCtx}_{foo_0}, \mathit{CallCtx}_{foo_1}, \mathit{CallCtx}_{bar} : \forall X_{main} :} \\
\quad \mathit{Sum}_{main}((x_0, g_0), (g_5)) \wedge & \\
\quad \mathit{CallCtx}_{foo_0}((x_0, g_1), (x_1, g_2)) \wedge & \\
\quad \mathit{CallCtx}_{foo_1}((x_1, g_2), (x_2, g_3)) \wedge & \\
\quad \mathit{CallCtx}_{bar}((x_2, g_3), (g_4)) \Longrightarrow & (\neg g_5 \vee \neg \mathit{true}) \wedge \\
& g_1 = (g_0 \wedge (x_0 < 10)) \wedge \\
& g_5 = ((g_0 \wedge \neg(x_0 < 10)) \vee g_4) \wedge \\
& \neg g_1 \wedge \neg g_2 \wedge \neg g_3
\end{array}$$

We obtain the following solutions for the predicates: $\mathit{CallCtx}_{bar} = \neg g_4$, $\mathit{CallCtx}_{foo_1} = \neg g_3$, $\mathit{CallCtx}_{foo_0} = \neg g_2$, $\mathit{Sum}_{main} = \neg g_0 \wedge \neg g_5$.

Then we recur into *bar* with (5) instantiated as:

$$\begin{array}{ll}
\multicolumn{2}{l}{\max \mathit{Sum}_{bar} : \forall z, g_8, g_9 :} \\
\quad \mathit{Sum}_{bar}((z, g_8), (g_9)) \Longrightarrow & (\neg g_9 \vee \neg(g_8 \Rightarrow z > 10)) \wedge \\
& \quad (g_9 = (g_8 \wedge z > 10))
\end{array}$$

Hence, we get for Sum_{bar}: $(g_8 \Rightarrow \neg(z > 10)) \wedge \neg g_9$.

In Line 6 of Algorithm 2', (5) for *main* is then:

$$\begin{array}{ll}
\multicolumn{2}{l}{\max \mathit{Sum}_{main}, \mathit{CallCtx}_{foo_0}, \mathit{CallCtx}_{foo_1}, \mathit{CallCtx}_{bar} : \forall X_{main} :} \\
\quad \mathit{Sum}_{main}((x_0, g_0), (g_5)) \wedge & \\
\quad \mathit{CallCtx}_{foo_0}((x_0, g_1), (x_1, g_2)) \wedge & \\
\quad \mathit{CallCtx}_{foo_1}((x_1, g_2), (x_2, g_3)) \wedge & \\
\quad \mathit{CallCtx}_{bar}((x_2, g_3), (g_4)) \Longrightarrow & (\neg g_5 \vee \neg \mathit{true}) \wedge \\
& g_1 = (g_0 \wedge (x_0 < 10)) \wedge \\
& g_5 = ((g_0 \wedge \neg(x_0 < 10)) \vee g_4) \wedge \\
& \neg g_1 \wedge \neg g_2 \wedge \\
& (g_3 \Rightarrow \neg(x_2 > 10)) \wedge \neg g_4
\end{array}$$

which results in $\mathit{CallCtx}_{bar} = \neg g_4$, $\mathit{CallCtx}_{foo_1} = g_3 \Rightarrow \neg(x_2 > 10)$, $\mathit{CallCtx}_{foo_0} = \neg g_2$, $\mathit{Sum}_{main} = \neg g_5$. Hence, $\mathit{CallCtx}_{foo}$ is updated to $g_7 \Rightarrow \neg(r > 10)$.

In the next iteration of $\mathit{compose}(\mathit{main})$ we recur into foo_1 and solve:

$$\begin{array}{ll}
\multicolumn{2}{l}{\max \mathit{Sum}_{foo} : \forall y, g_6, r, g_7 :} \\
\quad \mathit{Sum}_{foo}((y, g_6), (r, g_7)) \Longrightarrow & ((g_7 \Rightarrow \neg(r > 10)) \vee \neg \mathit{true}) \wedge \\
& \quad (g_6 = g_7) \wedge (r = y + 1)
\end{array}$$

Thus, Sum_{foo} is updated to $(g_6 \Rightarrow \neg(r > 10) \wedge g_7) \wedge (r = y + 1)$.

Then in Line 6 in $\mathit{compose}(\mathit{main})$, we solve

$$\begin{array}{ll}
\multicolumn{2}{l}{\max \mathit{Sum}_{main}, \mathit{CallCtx}_{foo_0}, \mathit{CallCtx}_{foo_1}, \mathit{CallCtx}_{bar} : \forall X_{main} :} \\
\quad \mathit{Sum}_{main}((x_0, g_0), (g_5)) \wedge & \\
\quad \mathit{CallCtx}_{foo_0}((x_1, g_2)) \wedge & \\
\quad \mathit{CallCtx}_{foo_1}((x_2, g_3)) \wedge & \\
\quad \mathit{CallCtx}_{bar}((g_4)) \Longrightarrow & (\neg g_5 \vee \neg \mathit{true}) \wedge \\
& g_1 = (g_0 \wedge (x_0 < 10)) \wedge \\
& g_5 = ((g_0 \wedge \neg(x_0 < 10)) \vee g_4) \wedge \\
& (g_1 \Rightarrow \neg(x_1 > 10) \wedge g_2) \wedge (x_1 = x_0 + 1) \wedge \\
& (g_2 \Rightarrow \neg(x_2 > 10) \wedge g_3) \wedge (x_2 = x_1 + 1) \wedge \\
& (g_3 \Rightarrow \neg(x_2 > 10)) \wedge \neg g_4
\end{array}$$

which gives us $\mathit{Sum}_{main} = (g_0 \Rightarrow \neg(x_0 > 8)) \wedge \neg g_5$. The calling contexts $\mathit{CallCtx}_{bar} = \neg g_4$, $\mathit{CallCtx}_{foo_0} = g_2 \Rightarrow \neg(x_1 > 10)$, and $\mathit{CallCtx}_{foo_1} = g_3 \Rightarrow \neg(x_2 > 10)$ do not result in an update of the calling contexts for *foo* and *bar* (Line 8 in Algorithm 2). $g_0 \wedge \mathit{Sum}_{main}$ is satisfiable, hence, $x \leq 8$ is a (maximal) refutation witness.

5 Examples of Refutation Algorithms

Algorithm 2' is not only applicable to straight-line programs with multiple procedure invocations, it can still be used for programs with loops by introducing invariants into the formula for the modular subproblem (5). However, in general it is hard to solve the problems without using approximations by bounding the number of unwindings and/or using abstractions for computing the predicates involved.

In the previous section, we have described the elements necessary for compositional, horizontal refutation proofs. In this section, we will give three examples of algorithms that instantiate this framework (Algorithm 2'), which we have implemented to compare them experimentally in Sect. 6. We assume that loops have been unwound a finite number of times before application of these techniques. The difference in the following three techniques lies in the abstractions that are used to solve for Sum_f and $\mathit{CallCtx}_f$ in (5). We consider techniques that use constraint solving to find counterexamples.

5.1 Concrete Backward Interpretation

This technique is the one sketched in the example at the beginning of Sect. 3. Formally, we use the domain of predicates that track a single constant value for each variable, defined as follows: Let $P(\boldsymbol{x}) = \{\mathit{false}\} \cup \{\boldsymbol{x} = \boldsymbol{d} \mid d_i \in \mathit{Dom}(x_i)\}$ with the domain $\mathit{Dom}(x_i)$ of variable x_i, then we admit the following predicates for summaries and calling contexts: $\mathit{Sum}_f \in \{g_f^{in} \Rightarrow p \mid p \in P(\boldsymbol{x}_f^{in})\}$ and $\mathit{CallCtx}_f \in \{g_f^{out} \Rightarrow p \mid p \in P(\boldsymbol{x}_f^{out})\}$. We explain now in an example how Algorithm 2' proceeds using this domain.

Example. Let us consider the example in Fig. 1 in Sect. 2. We start with $\mathit{compose}'(\mathit{main}, \neg g_5)$. We obtain the calling contexts $\neg g_2, \neg g_3, \neg g_4$ for foo_0, foo_1, bar, respectively. We recur into $\mathit{compose}'(\mathit{bar}, \neg g_9)$. We have to solve (5) where $\sum_{bar}$ is instantiated with the above domain:

$$\begin{array}{l} \exists d : \forall z, g_8, g_9 : \\ \quad (g_8 \Rightarrow z{=}d) \Longrightarrow (\neg g_9 \vee \neg(g_8 \Rightarrow (z > 10)))\wedge \\ \qquad\qquad\qquad\qquad (g_9 = (g_8 \wedge z > 10)) \end{array} \tag{6}$$

The partial order of our domain has only two levels *false* and the values for $\boldsymbol{d}$. Hence, we can implement max by $\exists \boldsymbol{d}$; if there is no $\boldsymbol{d}$ then $p = \mathit{false}$. A constraint solver may return, for example, $d = -4$; Sum_{bar} is hence $g_8 \Rightarrow (z = -4)$. This is an under-approximative summary of *bar* w.r.t. property violation.

In the next iteration of $\mathit{compose}'(\mathit{main}, \neg g_5)$ we solve:

$$\begin{array}{ll} \multicolumn{2}{l}{\exists d_0, \ldots, d_3 : \forall x_0, g_0, \ldots, g_5 :} \\ (g_0 \Rightarrow x_0{=}d_0)\wedge & \\ (g_2 \Rightarrow x_1{=}d_1)\wedge & \\ (g_3 \Rightarrow x_2{=}d_2)\wedge & \\ (g_4 \Rightarrow d_3) \Longrightarrow & (\neg g_5 \vee \neg \mathit{true})\wedge \\ & g_1 = (g_0 \wedge (x_0 < 10))\wedge \\ & g_5 = ((g_0 \wedge \neg(x_0 < 10)) \vee g_4)\wedge \\ & \neg g_1 \wedge \neg g_2 \wedge \\ & (g_3 \Rightarrow (x_2 = -4)) \wedge \neg g_5 \end{array} \tag{7}$$

and obtain $\mathit{CallCtx}_{\mathit{foo}_1} = (g_3 \Rightarrow (x_2 = -4))$. $\mathit{compose}'(\mathit{foo}, g_7 \Rightarrow (r = -4))$ returns $g_6 \Rightarrow (y = -5)$ for $\mathit{Sum}_{\mathit{foo}_1}$. Note that the boolean variable d_3 stands for the reachability of the exit of *bar*. Since *bar* has no return value, this is how its exit is encoded. Proceeding similarly we get $\mathit{compose}'(\mathit{foo}, g_7 \Rightarrow (r = -5)) = (g_6 \Rightarrow (y = -6))$; and finally $\mathit{Sum}_{\mathit{main}} = (g_0 \Rightarrow x_0 = -6)$. Hence, we have found a true global counterexample.

5.2 Abstract Backward Interpretation

Abstract backward interpretation computes sufficient preconditions to safety violations, i.e. negations of necessary preconditions to safety. The size of the summaries can vary from very concise to larger than the procedure, depending on the abstraction.

There are a couple of techniques to implement such abstract interpretations that are distinguished by the way abstract preconditions are inferred, e.g. (classical) abstract domain transformers (e.g. [20]), template-based synthesis (e.g. [15]) or interpolation (e.g. [1]).

We are going to use the template-based synthesis technique used in [3] to solve (5). We know how to compute over-approximative abstractions with that technique. Hence, we use an over-approximation to compute an under-approximation (similar to computing $\max f$ by $-\min(-f)$). This means we compute predicates $\mathit{Sum}_f^{\tilde{u}}$ and $\mathit{CallCtx}_j^{\tilde{u}}$ whose negations are Sum_f and $\mathit{CallCtx}_j$, respectively. This is done by solving the following formula in place of (5) in Algorithm 2'.

$$\begin{array}{l}
\min \mathit{Sum}_f^{\tilde{u}}, \overbrace{\mathit{CallCtx}_j^{\tilde{u}}, \ldots}^{\text{for all } j \in CS_f} : \forall X_f : \\
\big(\mathit{CallCtx}_f^{\tilde{u}}(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out}) \wedge \mathit{Sums}_f^{\tilde{u}} \wedge \\
T_f(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out}) \wedge \mathit{Props}_f(\boldsymbol{x}_f) \Longrightarrow \mathit{Sum}_f^{\tilde{u}}(\boldsymbol{x}_f^{in}, \boldsymbol{x}_f^{out}) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \bigwedge_{j \in CS_f} \mathit{CallCtx}_j^{\tilde{u}}(\boldsymbol{x}_j^{p\text{-}in}, \boldsymbol{x}_j^{p\text{-}out})\big)
\end{array} \tag{8}$$

This formula is derived from (5) by negating $(\mathit{CallCtx}_f \vee \neg \mathit{Props})$ on the right-hand side of (5), which yields $(\mathit{CallCtx}_f^{\tilde{u}} \wedge \mathit{Props})$, reversing the implication, and minimising to obtain an over-approximation for $\mathit{Sum}^{\tilde{u}}$ and $\mathit{CallCtx}^{\tilde{u}}$. Similar approaches are used in [5,10]. Since convex domains are too imprecise for this purpose, we use a disjunctive domain [22]. For our experiments we used intervals as a base domain. Formally, let $P(\boldsymbol{x}) = \{\bigvee_k \boldsymbol{d}'_k \leq \boldsymbol{x} \leq \boldsymbol{d}_k \mid d_i, d'_i \in \mathit{Dom}(x_i), k \geq 0\}$, then $\mathit{Sum}_f \in \{g_f^{in} \Rightarrow p \mid p \in P(\boldsymbol{x}_f^{in})\}$ and $\mathit{CallCtx}_f \in \{g_f^{out} \Rightarrow p \mid p \in P(\boldsymbol{x}_f^{out})\}$. Our implementation also ensures that arithmetic overflows create new disjuncts in order to avoid precision loss. The second source of additional disjuncts that we take into account are Lines 7 and 12 in Algorithm 2'.

Example. For the example in Fig. 1, we compute $\mathit{compose}'(\mathit{main}, \neg g_5)$. We solve (8) with $\mathit{CallCtx}_{\mathit{main}}^{\tilde{u}} = g_5$ and get $\mathit{CallCtx}_{\mathit{bar}}^{\tilde{u}} = g_4$, i.e. $\mathit{CallCtx}_{\mathit{bar}} = \neg g_4$.

We recur into $compose'(bar, \neg g_9)$, i.e. $CallCtx^{\tilde{u}}_{bar} = g_9$ We have to solve (8) instantiated with our domain.

$$\begin{array}{l} \exists d, d' : \forall z, g_8, g_9 : \\ \quad (g_9 \wedge true \wedge \\ \quad (g_9 = (g_8 \wedge z > 10)) \wedge (g_8 \Rightarrow (z > 10)) \Longrightarrow (g_8 \Rightarrow (d \leq z \leq d'))) \end{array} \tag{9}$$

Note that $Sums^{\tilde{u}}_f$ is true because the initial under-approximations are false—the superscript $\tilde{u}$ flags predicates that carry negations of under-approximations. We get $Sum^{\tilde{u}}_{bar} = (g_8 \Rightarrow (11 \leq z \leq MAX))$, i.e. $Sum_{bar} = (g_8 \wedge (MIN \leq z \leq 10))$. MAX and MIN denote the maximum, resp. minimum, possible value for the type of z.

We proceed similarly. Finally, $compose'(main, \neg g_5)$ computes $Sum^{\tilde{u}}_{main} = (g_0 \Rightarrow (9 \leq x_0 \leq MAX))$, i.e. $Sum_{main} = (g_0 \wedge (MIN \leq x_0 \leq 8))$.

Note that (8) expresses an over-approximation of *good* states; the complement is therefore guaranteed not to contain any good states, but only *bad* and *unreachable* states, and hence no strict under-approximation of bad states. However, this does not matter since we project $Sum_{f_{entry}}$ on the initial condition (see Theorem 2) to obtain a true under-approximation of inputs that violate a property.

Abstract backward interpretation is not limited to bounded unwindings of the transition relation, but can also be used for programs with loops (cf. [5,11]) by calling invariants into play in (8).

5.3 Symbolic Backward Interpretation

This technique computes the exact weakest precondition for the bounded problem. The technique is complete for loop-free programs. However, the size of the obtained summaries is in the same order as the procedure size in the worst case.

The domain used are sets of variables, so-called *dependency sets*. These sets of variables, X^{in}_f, X^{out}_f, $X^{p\text{-}in}_j$, $X^{p\text{-}out}_j$, describe which variables should be kept as relevant part of the summary. We then use them to compute an exact summary as the following predicate $Sum_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$:

$$\begin{array}{l} \exists X_f \setminus (X^{in}_f \cup X^{out}_f \cup \overbrace{X^{p\text{-}in}_j \cup X^{p\text{-}out}_j \cup \ldots}^{\text{for all } j \in CS_f}) : \\ \quad (CallCtx_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \vee \neg Props_f) \wedge T_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge Sums_f \end{array} \tag{10}$$

We implement the existential quantification in (10) by Gaussian elimination to eliminate as many of the intermediate or irrelevant variables as possible. After elimination the summary contains only variables that have a dependency on the property $Props_f$, on $\boldsymbol{x}^{out}$, or on the placeholder predicates, which are going to be replaced by summaries during the composition. The elimination can have positive and negative effects on the formula size depending on non-determinism and control flow paths in the procedure.

The composition operator is the same horizontal composition operator as in the two previous techniques. Context-sensitivity is exploited exactly in the same way as in the previous two techniques. The calling context at call site j is the set of output variables $X_j^{p\text{-}out}$ that a procedure call backward-transitively depends on the given property. The resulting calling context dependency set X_f^{out} is then used for eliminating intermediate variables in (10) in addition to the dependency sets obtained from $Sums_f$, and $Props_f$. The set of input variables X_f^{in} that have not been eliminated is the dependency set $X_f^{p\text{-}in}$ of the summary Sum_f.

Any satisfiable assignment to $\boldsymbol{x}_{f_{entry}}^{in}$ in the formula obtained by Gaussian elimination of the summary predicates in the entry function is a feasible global refutation.

Example. For example, in Fig. 1 the symbolic backward interpreter starts from the exit of *main* with $X_{main}^{out} = \emptyset$ to start with. As it arrives in *bar*, it retains the negation of the assertion $\neg(g_8 \Rightarrow (z > 10))$ and updates the dependency set to $X_{bar}^{in} = \{z, g_8\}$. On simplification, this gives the summary for *bar* as $g_8 \wedge \neg(z > 10)$.

Then the technique proceeds to the caller of *bar*, replacing the variables in the dependency set by the parameter passed, i.e. $X_{foo_1}^{p\text{-}out} = \{x_2, g_3\}$. Then it recurs into the call to *foo*. The statement $r = y + 1$ gives the summary of *foo* as $r = y + 1$ and the dependency set $\{y, g_6\}$. The next call to *foo* has already been analysed with the same dependency set, hence there is no need to recur.

Proceeding in the main function, we finally get the summary for *main* as $(g_1 = g_0 \wedge (x_0 < 0)) \wedge foo_0((x_0, g_1), (x_1, g_2)) \wedge foo_1((x_1, g_2), (x_2, g_3)) \wedge bar((x_2, g_3), (g_4))$. Substituting the placeholder predicates by their respective summaries (variables are renamed) allows us to evaluate the summary for *main*. Since it is satisfiable, we have found a global refutation.

6 Experiments

We performed a number of experiments to evaluate compositional refutation techniques in comparison with monolithic approaches.

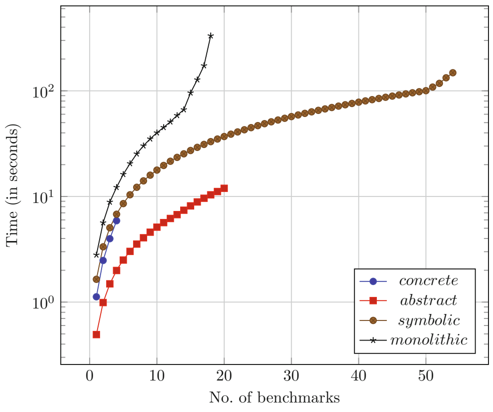

Fig. 2. Comparison on product-lines benchmarks

Implementation. We have implemented these safety refutation techniques as an extension to 2LS [3,24]. 2LS is a verification tool built on the CPROVER framework [9], using MiniSAT 2.2.1 as the backend solver (although other SAT and SMT solvers with incremental solving support can also be used). We limit resources to 900 s CPU time and 13 GB memory per benchmark. To aid reproducibility, we provide[4] the implementation sources along with the compilation instructions, the benchmarks, and scripts that can be used to run the tool on the benchmarks. As explained in Sect. 5, the three techniques are instances of a context-sensitive inter-procedural analysis that traverses the callgraph backwards and propagates summaries and calling contexts. For the concrete interpretation, values for non-deterministic choices are picked by the SAT solver. For the abstract interpretation we use disjunctions of intervals.

Benchmarks. We selected the unsafe examples (265 benchmarks) from the *product-lines* collection of the SV-COMP 2017 benchmarks set for our experiments. These benchmarks have a reasonably complex procedural structure (83 procedures per benchmark on average), which makes it suitable to test the effectiveness of our techniques. We set an unwinding depth of 5 for all the benchmarks, across all the techniques. The chosen depth might have been, in some cases, higher than what would be necessary to find a refutation. However, the aim of our experiments was to compare the scalability of the techniques in general, and not to find out the least amount of time needed to solve a given benchmark.

Results. Figure 2 shows the results plotting for each technique the cumulative time (y-axis) it takes to solve the given number of benchmarks (x-axis). The longer the line for a technique extends to the right the more benchmarks were solved within the resource limits. These results show some interesting tendencies. We observe that the symbolic backward interpretation performs best. It is complete, but could potentially degrade into a monolithic analysis if summaries cannot be sufficiently simplified and reused. But on this benchmark set it works quite well on a certain number of benchmarks. The abstract backward interpretation is very fast on a couple of benchmarks, but then remains inconclusive. This is supposedly due to the imprecision introduced by the weak abstract domain that we use. Yet, this is encouraging that by a clever choice of abstractions one could outperform the symbolic backward interpretation. The concrete backward interpretation succeeds only on very few benchmarks and is surprisingly slow. An explanation for this is that it is required to make non-deterministic choices that may turn out to be bad choices and make a counterexample infeasible. Moreover, the summaries that it computes usually do not generalise beyond the procedure invocation they were generated for. Hence, this technique is likely to degrade into following the entire execution path, spoiling the benefits of modularity while exhibiting the drawbacks of abstraction. The monolithic analysis (BMC), which is based on full inlining is slowest but solves almost as many benchmarks as the abstract one.

[4] https://github.com/kumarmadhukar/2ls/tree/atva17.

7 Conclusion

We investigated compositional refutation techniques in horizontal, e.g. procedure-modular, decompositions of sequential programs. We showed how to derive a compositional refutation framework step by step from the monolithic problem. We also compared the completeness properties of concrete, abstract and symbolic modular refutation approaches. Our experiments show that compositional refutation techniques have an advantage over monolithic approaches, however, not all tested approaches perform equally well because of their varying completeness. Using a portfolio of fast incomplete techniques and slower complete ones may ensure that modular techniques are always at least as fast as monolithic ones in practice.

Open Questions. Modular analyses should be independent of a program's syntactic structure because real-world programs are not written in a nice and balanced way that would enable efficient modular analysis. It would be worthwhile to explore semantic decompositions into modules in order to make these techniques scale on real-world programs. W.r.t. the inter-procedural backward analysis, it remains to be investigated how to handle recursion.

Moreover, it would be interesting to look into compositional refutation in termination analysis. Also there, spuriousness of local refutations can occur due to lack of context information: To find a counterexample to termination one needs to find a stem from the entry point. Compositionality in this context has been explored in the Ultimate tool [17]. We would also consider performance comparisons with testing, i.e. dynamic refutation techniques (random, directed, concolic, etc.) to be beneficial to advance research in static refutation techniques.

Related Work. Compositional automated verification approaches have been considered in the tools Whale [1] and FunFrog [26], for example. Horn clause encodings were used in [18]. These tools eventually use interpolation to compute abstractions. Under-approximating precondition inference techniques have been proposed for polyhedra [20] and with the help of bit blasting and loop iteration estimation [4]. All these techniques can be used in our setting, however, their completeness properties remain to be evaluated. Completeness considerations [21] have been conducted for compositional LTL model checking [7,8] of (parallel) compositions of (infinite-) state transition systems. Since the decomposition of sequential programs can be encoded into a composition of parallel programs (with appropriate synchronisation), their completeness results are expected to hold in our setting. Compositionality has also been explored in the context of dynamic test generation to achieve scalability by memoizing symbolic execution sub-paths as test summaries [13]. This has given rise to an incremental approach for statically validating symbolic test summaries against code changes [14]. In our framework memoization is naturally handled by the composition operator.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Whale: an interpolation-based algorithm for inter-procedural verification. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 39–55. Springer, Heidelberg (2012). doi:10.1007/978-3-642-27940-9_4
2. Alur, R., de Alfaro, L., Henzinger, T.A., Mang, F.Y.C.: Automating modular verification. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 82–97. Springer, Heidelberg (1999). doi:10.1007/3-540-48320-9_8
3. Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 145–161. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48288-9_9
4. Brauer, J., Simon, A.: Inferring definite counterexamples through under-approximation. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 54–69. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28891-3_7
5. Chen, H., David, C., Kroening, D., Schrammel, P., Wachter, B.: Synthesising interprocedural bit-precise termination proofs. In: Automated Software Engineering, pp. 53–64. ACM (2015)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). doi:10.1007/10722167_15
7. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. Trans. Programm. Lang. Syst. **16**(5), 1512–1542 (1994)
8. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: Logic in Computer Science, pp. 353–362. IEEE Computer Society (1989)
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24730-2_15
10. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-70545-1_32
11. David, C., Kesseli, P., Kroening, D., Lewis, M.: Danger invariants. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 182–198. Springer, Cham (2016). doi:10.1007/978-3-319-48989-6_12
12. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Programming Language Design and Implementation, pp. 234–245. ACM (2002)
13. Godefroid, P.: Compositional dynamic test generation. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, pp. 47–54. ACM, New York (2007)
14. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically validating must summaries for incremental compositional dynamic test generation. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 112–128. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23702-7_12
15. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Programming Language Design and Implementation, pp. 281–292. ACM (2008)
16. Harman, M., Hierons, R.M.: An overview of program slicing. Softw. Focus **2**(3), 85–92 (2001)

17. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. Comput.-Aided Verification **8559**, 797–813 (2014)
18. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: Formal Methods in Computer-Aided Design, pp. 89–96 (2015)
19. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17511-4_20
20. Miné, A.: Inferring sufficient conditions with backward polyhedral under-approximations. Electr. Notes Theor. Comput. Sci. **287**, 89–100 (2012)
21. Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning methods. ACM Trans. Comput. Log. **11**(3), 16:1–16:22 (2010). doi:10.1145/1740582.1740584
22. Sankaranarayanan, S., Ivančić, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 3–17. Springer, Heidelberg (2006). doi:10.1007/11823230_2
23. Schrammel, P.: Challenges in decomposing encodings of verification problems. In: Horn Clauses for Verification and Synthesis, EPTCS (2016). p. to appear
24. Schrammel, P., Kroening, D.: 2LS for program analysis. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 905–907. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_56
25. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Successful use of incremental BMC in the automotive industry. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 62–77. Springer, Cham (2015). doi:10.1007/978-3-319-19458-5_5
26. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 160–175. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34188-5_15
27. SPARK: (2014). http://www.spark-2014.org/