Counterexample-Guided Precondition Inference

Counterexample-guided Precondition Inference[⋆]

Mohamed Nassim Seghir and Daniel Kroening

Computer Science Department, University of Oxford

Abstract. The precondition for an assertion within a procedure is useful for understanding, verifying and debugging programs. As the procedure might be used in multiple calling-contexts within the program, the precondition should be sufficiently precise to enable re-use. We present an extension of counterexample-guided abstraction refinement (CEGAR) for automated precondition inference. Starting with an overapproximation of both the set of safe and unsafe states, we iteratively refine them until they become disjoint. The resulting precondition is then necessary and sufficient for the validity of the assertion, which prevents false alarms. We have implemented our approach and present experimental results using string and array-manipulating programs.

1 Introduction

Software model checking is a popular technique for program verification. A diverse range of tools based on this approach have been developed (e.g., SLAM [1], BLAST [19], MAGIC [7], SATABS [8] and TERMINATOR [11]) and successfully applied to real-world software. The key to effectiveness of these tools is *abstraction*, and predicate abstraction [16] is a well-established instance. The predicate discovery in tools implementing it is driven by *counterexample-guided abstraction refinement* [9], commonly known as CEGAR.

Most of the tools above answer the usual verification question: "given an assertion at some program location, is this assertion always valid?" When considering just a fragment of a program containing an assertion, we can ask a slightly different question: "In which context is the assertion valid?" The code fragment might be a procedure that is called at different program locations, hence the computed context should be as general as possible to be reusable at the different call sites. A simple and straightforward way to infer a precondition is to compute a conservative abstraction of the set of unsafe states, i.e., those states that can reach an error, and using its complement as precondition. The problem with this approach is that an over-approximation of the set of unsafe states might include safe states as well, resulting in an over-conservative precondition. The abstraction must then be refined by removing some of the safe states. This cannot be performed in an enumerative fashion, as the set of safe states is often infinite.

[⋆] Supported by ERC project 280053, EPSRC project EP/H017585/1, the EU FP7 STREP PINCETTE and the ARTEMIS VETESS project.

We propose a solution to the problem based on the abstraction (and thus generalization) of both the set of safe and unsafe states. Our approach is based on the CEGAR paradigm: starting with an over-approximation of both sets, we iteratively refine them until they become disjoint. Thus, the resulting precondition is sufficient and also necessary for the validity of the assertion. This guarantees the absence of false alarms, as the violation of the precondition by some calling context entails the violation of the assertion within the procedure. Our contributions are summarized as follows:

- A novel approach to generate *exact* preconditions, i.e., necessary and sufficient. Thus, the precondition is independent from the calling context. Most of the approaches in the literature generate preconditions that are only sufficient, thus the precondition often has to be re-adjusted if it is not satisfied by some calling context. In our case, a violation of the precondition will result in a real error, and we thus avoid false alarms.
- An implementation of the approach using ingredients that are common to most CEGAR-based verification tools. Thus, our technique represents a generic scheme for extending other tools to infer preconditions.
- A simple predicate inference mechanism for algorithms that manipulate arrays used on the top of the standard predicate refinement procedure. This simple technique generates predicates that are often adequate to obtain the right program invariant and subsequently obtain the desired precondition.

The remainder of this paper is organized as follows: Section 2 illustrates our approach by means of examples. Section 3 introduces background material. Section 4 describes our approach for precondition inference and the refinement technique used in the CEGAR loop. Section 5 presents experimental results and Section 6 discusses related work.

2 Examples

Consider the program copy given in Figure 1(a). It takes as parameters two arrays a and b and the length b_l of array b. The program copies the elements of array b in the range $\{0, \ldots, b\_l - 1\}$ to the corresponding range in array a. The access to array a is safe if the index expression is in the range $\{0, \ldots, a\_l - 1\}$, where a_l is the length of array a. It is trivial to see that the lower bound is not violated. Let us then focus on the upper bound. The safety condition with regards to the upper bound is expressed by the assertion at location ℓ_2. Our goal is to find a precondition for procedure copy that guarantees that this assertion is never violated. The precondition must be expressed only using program elements visible at the entry-point of the procedure, i.e., it must be a predicate over the procedure parameters and the global variables. The precondition should also be *exact*, i.e., it should neither be too strong nor too weak.

Transformation to reachability We will now illustrate our approach to precondition inference approach. We use standard notation and formally represent programs in terms of transition constraints over primed and unprimed program

```
void copy(int a[], int b[], int b_l)
{
     int i;
ℓ0 : i = 0;
ℓ1 : while(i < b_l)
     {
ℓ2 :      assert(i < a_l);
          a[i] = b[i];
          i++;
     }
}
```

(a)

```
void copy_2(int a[], int b[])
{
     int i;
ℓ0 : i = 0;
ℓ1 : while(b[i] != 0)
     {
ℓ2 :      assert(i < a_l);
          a[i] = b[i];
          i++;
     }
}
```

(b)

Fig. 1. Two simple programs that copy a range of elements from array b to array a. In procedure copy, the limit of the range to be copied is explicitly given via b_l. In copy_2, the range is implicitly delimited via the sentinel value 0.

variables. The set of transition constraints corresponding to program copy (Figure 1(a)) is given in Figure 2(a) and the associated control flow graph is given in Figure 2(b). The program counter is modeled explicitly using the variable pc, which ranges over the set of control locations. The assertion in the original program is replaced with a conditional branch whose condition is the negation of the assertion and whose target is the error location ℓ_E. The special location ℓ_F is the *final location*, and has no successor.

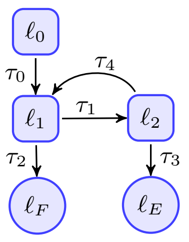
Observe that the error location is only reachable if $i \geq a\_l$ evaluates to true at location ℓ_2. The final location ℓ_F is reached in paths without error. The transition τ_0 corresponds to the initialization of variable i. The transition τ_1 represents the entrance to the loop and τ_2 the exit from the loop. The assertion is modeled via the transition τ_3, which conditionally alters the control flow to the error location. Finally, the transition τ_4 models the remainder of the loop body after the assert statement. Arrays a and b are represented by uninterpreted function symbols, and $a[x := e]$ denotes function update (the expression is equal to a where the x^{th} element has been replaced by e).

Over-approximating the unsafe states It is in general not possible to enumerate all the traces of a program. In our example, the program contains a cycle $\langle\tau_1; \tau_4\rangle$ (Figure 2(b)) that can be unfolded an indefinite number of times, leading to an infinite number of traces. A solution to this problem is to provide a *backwards inductive invariant*: an invariant that includes all error states and which is inductive under the application of wp[1]. Predicate abstraction [16] is a suit-

[1] $\mathsf{wp}(\tau, \varphi)$ is the weakest precondition for the formula φ with respect to statement (transition constraint) τ. It extends to a sequence of statements (trace) π.

$$\begin{aligned}
\tau_0 :&\ pc = \ell_0 \wedge i' = 0 \wedge pc' = \ell_1\\
\tau_1 :&\ pc = \ell_1 \wedge i < b\_l \wedge i' = i \wedge pc' = \ell_2\\
\tau_2 :&\ pc = \ell_1 \wedge i \geq b\_l \wedge i' = i \wedge pc' = \ell_F\\
\tau_3 :&\ pc = \ell_2 \wedge i \geq a\_l \wedge i' = i \wedge pc' = \ell_E\\
\tau_4 :&\ pc = \ell_2 \wedge i < a\_l \wedge a' = a[i := b(i)]\\
&\ \wedge i' = i + 1 \wedge pc' = \ell_1
\end{aligned}$$

(a)



(b)

Fig. 2. Transition constraints for program copy (a) and the corresponding graphical representation (b) or the control flow graph

able technique for building such an invariant. The key challenge when applying predicate abstraction is the choice of predicates.

Naive approaches for inferring predicates, e.g., based on weakest preconditions, often diverge [21]. In our example, suppose that we first obtain the error path $\tau_0; \tau_1; \tau_3$ from the abstract model. The analysis of this path via wp, i.e., $\mathsf{wp}(\tau_0; \tau_1; \tau_3, pc = \ell_E)$, gives us the formula $0 < b\_l \wedge 0 \geq a\_l$. We add the predicates $0 < b\_l$ and $0 \geq a\_l$ and set the precondition φ to be $0 \geq b\_l \vee 0 < a\_l$. If we unfold the loop once more we obtain the error trace $\tau_0; \tau_1; \tau_4; \tau_1; \tau_3$. We have $\mathsf{wp}(\tau_0; \tau_1; \tau_4; \tau_1; \tau_3, pc = \ell_E) \equiv 1 < b\_l \wedge 1 \geq a\_l$. This new trace is not covered by the previous one, as it is still feasible under our precondition φ. We then update φ to rule out the new trace to obtain $(0 \geq b\_l \vee 0 < a\_l) \wedge (1 \geq b\_l \vee 1 < a\_l)$. After unfolding the loop j times, we obtain $\mathsf{wp}(\tau_0; \langle\tau_1; \tau_4\rangle^j; \tau_1; \tau_3, pc = \ell_E) \equiv j < b\_l \wedge j \geq a\_l$ and the precondition

$$\varphi \equiv \bigwedge_{j>0} j \geq b\_l \vee j < a\_l\ .$$

We can continue to unfold the loop, every time generating a new trace that is not covered by the previous ones. To address this divergence, we go beyond the syntactic approach to predicate discovery and use techniques to infer more general facts. For example, by linearly combining the predicates $0 < b\_l$ and $0 \geq a\_l$ (from the first iteration) we deduce $a\_l < b\_l$. This new predicate is a backwards invariant at location ℓ_0 with respect to the program, i.e.,

$$(\bigvee_{j>0} \mathsf{wp}(\pi_j, pc = \ell_E)) \ \Rightarrow\ (pc = \ell_0 \Rightarrow a\_l < b\_l)\ .$$

Thus, the predicate $a\_l < b\_l$ over-approximates the set of states that reach the error location. The precondition φ is then simply chosen to be $a\_l \geq b\_l$, i.e., the complement of that set.

Over-approximating the safe states When over-approximating the set of states that reach the error location, we always include all error states but may include some safe entry states as well. It means that the precondition, which is the complement of the computed set, may exclude safe traces and is thus unnecessarily strong. To tune the precision of the abstracted set of error states, our new algorithm also over-approximates the set of entry states that reach the final location ℓ_F (the safe states). We then check the intersection of this set with the (over-approximation of the) states that reach the error location. If the intersection is empty, we conclude that our current set of unsafe states does not include any safe state.

As we did for the error location, we obtain the over-approximation of the set of states reaching the final location given as the state formula $a\_l \geq b\_l \vee 0 \geq b\_l$. The intersection of this set with the set of unsafe states ($a\_l < b\_l$) is obtained by forming the conjunction:

$$(a\_l \geq b\_l \vee 0 \geq b\_l) \wedge (a\_l < b\_l)$$

The formula above has satisfying assignments, which means that the two sets are not disjoint. As $a\_l \geq b\_l$ and $a\_l < b\_l$ are inconsistent, the intersection can only be in $0 \geq b\_l$. Thus, from $0 \geq b\_l$ we can reach both the error and final location. This outcome is caused by insufficient precision of the abstraction. Let us consider two traces, $\pi_E = \tau_0; \tau_1; \tau_3$ leading to the error location, and $\pi_F = \tau_0; \tau_2$ leading to the final location. We have $\mathsf{wp}(\pi_E, pc = \ell_E) \equiv 0 < b\_l \wedge 0 \geq a\_l$ and $\mathsf{wp}(\pi_F, pc = \ell_F) \equiv 0 \geq b\_l$. Thus, π_E is not feasible from states with $0 \geq b\_l$, which means that the set of unsafe states is not precise enough. It is then refined by adding the predicate $b\_l > 0$, which makes the two sets disjoint. The final precondition φ is given by

$$0 \geq b\_l \vee a\_l \geq b\_l\ .$$

The precondition φ is now necessary and sufficient, meaning that is does not allow any state to reach the error location and does not exclude any state that reaches the final location.

Inferring quantified preconditions Let us consider a slightly modified version of the previous program copy, which is given in Figure 1(b). In this example, the range of elements to be copied from array b to a is not explicit, as it is indicated via a sentinel value (0 in the example). After going through the different steps described for the previous example, our method succeeds in inferring the precondition

$$(b[0] = 0) \vee (\exists x \in \{0, \ldots, a\_l\}.\ b[x] = 0)\ .$$

Observe that the precondition inferred by the algorithm is not equivalent to $\exists x \in \{0, \ldots, a\_l\}.\ b[x] = 0$; it is weaker. This is due the possibility of skipping the loop when $b[i] \neq 0$ is false, regardless of the value of a_l. This implies that runs from states in which $b[0] = 0$ are safe. This case is expressed via the first disjunct of the precondition above. We will re-visit the second example in Section 4 to illustrate our refinement procedure.

3 Preliminaries

In this section, we provide background on counterexample-guided abstraction refinement for predicate abstractions.

Program To aid the formal presentation, we assume that a program is given as set $\mathcal{TC}$ of transition constraints τ. A transition constraint τ is a formula of the form

$$g(X) \wedge x'_1 = e_1(X) \wedge \ldots \wedge x'_n = e_n(X) \tag{1}$$

where $X = \langle x_1, \ldots, x_n \rangle$ is a tuple (vector) of program variables, which include the program counter *pc*. In (1), unprimed variables refer to the program state before performing the transition and primed ones represent the program state after performing the transition. Formula $g(X)$ is called the *guard* and the remaining conjuncts of τ are the *update* or *assignment.*

Representing states symbolically Let us write $\mathcal{V} = \{x_1, \ldots, x_n\}$ for the set of variables of the program (including the program counter *pc*). For a variable $x \in \mathcal{V}$, $\mathit{Type}(x)$ is the type (range) of x and $\sigma(x)$ is a valuation of x such that $\sigma(x) \in \mathit{Type}(x)$. The variable *pc* ranges over the set of all program locations. Given X (a tuple of the variables), a program state is the valuation $\sigma(X) = \langle \sigma(x_1), \ldots, \sigma(x_n) \rangle$.

A set of program states S is represented symbolically by means of the *characteristic function* of S. The formula φ represents the set of all those states that correspond to a satisfying assignment of φ, i.e., $\{\sigma(X) \mid \varphi(X)\}$. We will use sets and their characteristic functions interchangeably. Symbolic states (formulas) are partially ordered via the implication operator $\Rightarrow$, i.e., $\varphi' \leq \varphi$ means $\varphi' \Rightarrow \varphi$.

State transformer For a formula φ, the application of the operator pre with respect to the transition constraint τ returns a formula representing the set of all predecessor states of φ under the transition constraint τ, formally

$$\mathsf{pre}(\tau, \varphi(X)) \equiv g(X) \wedge \varphi[\langle e_1(X), \ldots, e_n(X) \rangle / X] \ .^2$$

For the whole program $\mathcal{TC}$, pre is given by

$$\mathsf{pre}(\varphi(X)) \equiv \bigvee_{\tau \in \mathcal{TC}} \mathsf{pre}(\tau, \varphi(X)) \ .$$

For a trace $\pi = \tau_1; \ldots; \tau_n$, we have

$$\mathsf{pre}(\tau_1; \ldots; \tau_n, \varphi) = \mathsf{pre}(\tau_1, \ldots \mathsf{pre}(\tau_{n-1}, \mathsf{pre}(\tau_n, \varphi))) \ .$$

If $\mathsf{pre}(\pi, \varphi)$ is not equivalent to false, then the trace π is *feasible.*

[2] The notation $f[Y/X]$ represents the expression obtained by replacing all occurrences of every variable from the vector X in f with the corresponding variable from Y. It naturally extends to a collection (set or list) of expressions.

(Un)Safe states To ease presentation, let us assume that the program contains a single error location ℓ_E and a single final location ℓ_F ($\ell_E \neq \ell_F$).[3] We denote by bad the set of *error states*, which is simply given by $pc = \ell_E$. Similarly, we call final the set of *final states*, which is represented by $pc = \ell_F$.

The set of safe states safe contains all states from which a final state is reachable. Formally,

$$\mathsf{safe} \equiv \mathsf{lfp}(\mathsf{pre}, \mathsf{final}) \tag{2}$$

where $\mathsf{lfp}(\mathsf{pre}, \varphi)$ denotes the least fixpoint of the operator pre above φ. Similarly, unsafe is the set of all states from which an error (bad) state is reachable:

$$\mathsf{unsafe} \equiv \mathsf{lfp}(\mathsf{pre}, \mathsf{bad}) \ . \tag{3}$$

The least fixpoints represent inductive backwards invariants, which we denote by ψ_{bad} and ψ_{final}, respectively. The invariants are inductive under pre, i.e.,

– $\mathsf{bad} \leq \psi_{\mathsf{bad}}$ and $\mathsf{final} \leq \psi_{\mathsf{final}}$
– $\mathsf{pre}(\psi_{\mathsf{bad}}) \leq \psi_{\mathsf{bad}}$ and $\mathsf{pre}(\psi_{\mathsf{final}}) \leq \psi_{\mathsf{final}}$

In the absence of non-determinism in the program, the sets of unsafe and safe states are disjoint, and we have

$$\mathsf{unsafe} \wedge \mathsf{safe} \equiv \mathsf{false} \ .$$

Predicate abstraction Predicate abstraction consists of approximating a state φ with a formula φ' constructed as a Boolean combination of predicates taken from a set P. Here, the term approximation means that any model that satisfies φ must satisfy φ'. Thus, a suitable approximation is obtained via the logical implication "$\Rightarrow$", i.e., φ' is the strongest Boolean combination built up from predicates taken from the finite set P such that $\varphi \Rightarrow \varphi'$.

Defining the abstraction function α as being the strongest Boolean combination of predicates in P is not practical because of the exponential complexity of the problem. Therefore, we use a lightweight version of α that consists of building the strongest conjunction of predicates in P:

$$\alpha(\varphi) \equiv \bigwedge p \quad | \quad p \in P \wedge \varphi \Rightarrow p \ .$$

Let us have $\mathcal{D}^\sharp$ the domain of formulas built up from the finite set of predicates P. The domain $\mathcal{D}^\sharp$ is not closed under pre, therefore, we define $\mathsf{pre}^\sharp$ under which $\mathcal{D}^\sharp$ is closed. Let us associate the concretization function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ to α, we simply choose γ to be the identity function. Functions α and γ form a *Galois connection* with respect to $\geq$ ($\Leftarrow$) being the partial order relation for both $\mathcal{D}$ and $\mathcal{D}^\sharp$. Formally speaking

$$\forall x \in \mathcal{D}\ \forall y \in \mathcal{D}^\sharp.\ \alpha(x) \geq y \Leftrightarrow x \geq \gamma(y) \ .$$

[3] In case of multiple assertions, we add an edge from each assertion (guarded with the negation of the assertion) to ℓ_E. Similar treatment can be applied in the case of multiple return locations.

Hence, we define $\mathsf{pre}^\sharp : \mathcal{D}^\sharp \rightarrow \mathcal{D}^\sharp$, the abstract version of pre, as follows:

$$\mathsf{pre}^\sharp(\varphi) \equiv \alpha(\mathsf{pre}(\gamma(\varphi))) \ ,$$

and thus

$$\mathsf{pre}^\sharp(\tau, \varphi) = \alpha(\mathsf{pre}(\tau, \varphi)) = \bigwedge p \quad | \quad p \in P \wedge \mathsf{pre}(\tau, \varphi) \Rightarrow p \ .$$

As seen for pre, the operator $\mathsf{pre}^\sharp$ also extends to traces. Later on in this paper, whenever we write $\mathsf{pre}^\sharp_P$ we mean that the abstraction (image) is computed by considering predicates from the set P.

The lattice of abstract states $(\mathcal{L}, \Rightarrow)$ is finite as the set of predicates is finite. Therefore, $\mathsf{lfp}(\mathsf{pre}^\sharp, \mathsf{bad})$ $(\mathsf{lfp}(\mathsf{pre}^\sharp, \mathsf{final}))$, the least fixpoint for $\mathsf{pre}^\sharp$ above bad (final) in $\mathcal{L}$, is computable.

4 Precondition Inference

The precondition inference problem can be described as the computation of a formula φ such that:

$$\mathsf{lfp}(\mathsf{pre}, \mathsf{bad}) \wedge \varphi \equiv \mathsf{false} \tag{4}$$

The fixpoint for the preimage-operator pre is in general not computable, we thus compute the least fixpoint for $\mathsf{pre}^\sharp$. As we have $\mathsf{lfp}(\mathsf{pre}, \mathsf{bad}) \leq \mathsf{lfp}(\mathsf{pre}^\sharp, \mathsf{bad})$, it is sufficient to show that

$$\mathsf{lfp}(\mathsf{pre}^\sharp, \mathsf{bad}) \wedge \varphi \equiv \mathsf{false}$$

to conclude the validity of (4). The precondition φ can then be simply chosen as the negation of $\mathsf{lfp}(\mathsf{pre}^\sharp, \mathsf{bad})$ projected on the entry location. One problem with this approach is that due to the abstraction we may exclude some of the safe, terminating runs. A second challenge is the choice of predicates. We have seen in the example above that a bad choice of predicates can lead to divergence. In what follows, we present a new CEGAR-based algorithm for precondition inference that guarantees that all safe executions are included in the precondition. We also propose a predicate discovery mechanism that goes beyond the approach based on weakest preconditions.

4.1 Counterexample-guided precondition inference

Our goal is to increase the precision of the set of unsafe states $\mathsf{unsafe}^\sharp$, making it free of safe states. This is non-trivial, since we cannot enumerate safe states, as there are in general infinitely many. Hence, we need to construct the set of safe states by over-approximating them as well. Our idea consists of building abstractions of increasing precision of both the set of safe and unsafe states until they become disjoint. We propose an implementation of this idea by extending the classical CEGAR paradigm, where its main ingredients are instantiated in our setting with the following:

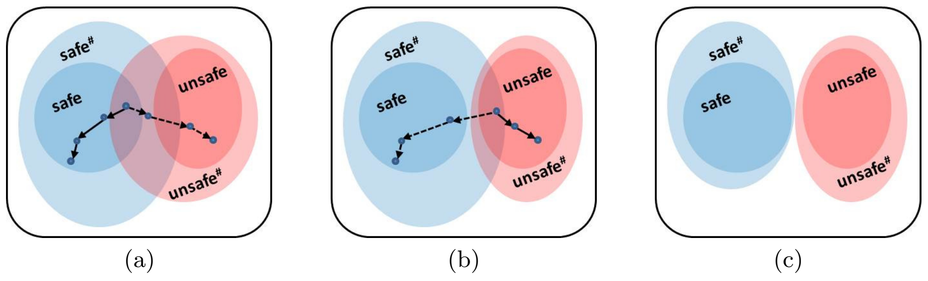


Fig. 3. Illustration of the main phases of algorithm InferPrecond. Dashed arrows indicate that the trace is spurious.

- **Abstraction**: we abstract both the set of safe and unsafe states.
- **Counterexample**: in our context, a counterexample is *two* abstract traces, a safe one and an unsafe one, beginning with a common initial state.
- **Counterexample simulation**: checks if the two traces can be concretized to effectively share a common concrete initial state. This is only possible in the presence of non-determinism in the program. The check is carried out by computing the weakest precondition for each trace. Hence, the counterexample is spurious if the two preconditions are disjoint.
- **Refinement**: the spurious counterexample is ruled out by adding predicates that refine the abstraction such that the two traces cannot share their initial state.
- **Termination criterion**: the iterative process stops when the two abstractions (of safe and unsafe states) are disjoint.

We present algorithm InferPrecond (Algorithm 1), which implements a counterexample-guided abstraction refinement loop for precondition inference. The algorithm starts with an over-approximation of both the set of safe and unsafe states (lines 5 and 6), denoted by $\mathsf{safe}^\sharp$ and $\mathsf{unsafe}^\sharp$, respectively. It iteratively refines them until their projections onto the initial location become disjoint, i.e., $(\mathsf{safe}^\sharp \wedge \mathsf{unsafe}^\sharp \wedge pc = \ell_0) \equiv \mathsf{false}$ (Figure 3(c)). The computed precondition is then the set of safe states projected onto the initial location ℓ_0 (line 8 of the algorithm).

The refinement process is applied whenever $\mathsf{safe}^\sharp$ and $\mathsf{unsafe}^\sharp$ intersect, i.e., when we have a bad trace and a safe one sharing their initial state. In Figure 3(a), $\mathsf{safe}^\sharp$ and $\mathsf{unsafe}^\sharp$ intersect, but the analysis reveals that the initial state is in reality in safe, thus the (dashed) trace in $\mathsf{unsafe}^\sharp$ is the one that is spurious. After refining $\mathsf{unsafe}^\sharp$, we obtain the abstraction in Figure 3(b). The two sets still intersect, however this time the spurious trace is in $\mathsf{safe}^\sharp$, as the initial state belongs to unsafe. The refinement process is carried out by calling the procedure Refine at line 14. This procedure takes as parameters two traces, one leading to the error location and another one leading to the final location, and returns a new set of predicates. We describe this procedure in detail in the next section.

Algorithm 1: InferPrecond

```
Input: set of transition constraints (program) $\mathcal{TC}$
Output: formula (precondition)
1  Var $P$: set of predicates;
2  Var $\mathsf{safe}^\sharp$, $\mathsf{unsafe}^\sharp$: formula;
3  $P := \emptyset$;
4  while true do
5      $\mathsf{unsafe}^\sharp := \mathsf{lfp}(\mathsf{pre}^\sharp_P, \mathsf{bad})$;
6      $\mathsf{safe}^\sharp := \mathsf{lfp}(\mathsf{pre}^\sharp_P, \mathsf{final})$;
7      if $(\mathsf{safe}^\sharp \wedge \mathsf{unsafe}^\sharp \wedge pc = \ell_0) \equiv \mathsf{false}$ then
8          return $(\mathsf{safe}^\sharp \wedge pc = \ell_0)$;
9      Let $\pi_E$ and $\pi_F$ two traces s.t. $\mathsf{pre}^\sharp_P(\pi_E, \mathsf{bad}) \wedge \mathsf{pre}^\sharp_P(\pi_F, \mathsf{final}) \not\equiv \mathsf{false}$;
10     if $\mathsf{pre}(\pi_E, \mathsf{bad}) \wedge \mathsf{pre}(\pi_F, \mathsf{final}) \not\equiv \mathsf{false}$ then
11         print("warning: non-determinism in program");
12         exit;
13     else
14         $P := P \cup \mathsf{Refine}(\pi_E, \pi_F)$;
```

Proposition 1. *The precondition φ computed by algorithm* InferPrecond *(a) guarantees the non-reachability of bad states and (b) the non-exclusion of safe terminating traces.*

Proof. **(a)** *φ guarantees non-reachability of bad states.* As computed by algorithm InferPrecond, $\varphi \equiv \mathsf{safe}^\sharp \wedge pc = \ell_0$. Let us assume that there are states in φ from which a bad state can be reached. Thus, there is an error trace π_E such that

$$\mathsf{pre}(\pi_E, \mathsf{bad}) \wedge \mathsf{safe}^\sharp \wedge pc = \ell_0 \not\equiv \mathsf{false} \tag{5}$$

We also know that

$$\mathsf{pre}(\pi_E, \mathsf{bad}) \Rightarrow \mathsf{unsafe}^\sharp \ , \tag{6}$$

as $\mathsf{lfp}(\mathsf{pre}, \mathsf{bad}) \leq \mathsf{lfp}(\mathsf{pre}^\sharp, \mathsf{bad})$. From (5) and (6) we obtain

$$\mathsf{unsafe}^\sharp \wedge \mathsf{safe}^\sharp \wedge pc = \ell_0 \not\equiv \mathsf{false} \ ,$$

which contradicts the return condition at line 7 of algorithm InferPrecond.

(b) *φ does not exclude safe terminating traces.* Let us assume that φ excludes a given safe terminating trace π_F from ℓ_0 to ℓ_F, which means that

$$\mathsf{pre}(\pi_F, \mathsf{final}) \wedge \mathsf{safe}^\sharp \equiv \mathsf{false}$$

or

$$\mathsf{pre}(\pi_F, \mathsf{final}) \Rightarrow \neg\mathsf{safe}^\sharp \ . \tag{7}$$

We also have

$$\mathsf{pre}(\pi_F, \mathsf{final}) \Rightarrow \mathsf{safe}^\sharp \ , \tag{8}$$

as $\mathsf{lfp}(\mathsf{pre}, \mathsf{final}) \leq \mathsf{lfp}(\mathsf{pre}^\sharp, \mathsf{final})$. From (7) and (8) we conclude $\mathsf{pre}(\pi_F, \mathsf{final}) \equiv \mathsf{false}$, which means that such a trace π_F is not feasible. □

As program model checking is not decidable, we have no guarantee for termination of algorithm InferPrecond. However, whenever it terminates, the above proposition holds.

4.2 Refinement for precondition inference

The main goal of refinement is to generate the minimal possible set of predicates that rule out a maximum number of spurious traces. Hence, the generated predicates must be as general as possible. We present procedure Refine, which takes as parameters two traces, one trace π_E leading to the the error location, and another one π_F leading to the final location. The returned result is a set of predicates P that enables the verifier to show the following:

$$\mathsf{pre}^\sharp_P(\pi_E, \mathsf{bad}) \wedge \mathsf{pre}^\sharp_P(\pi_F, \mathsf{final}) \equiv \mathsf{false} .$$

The procedure Refine relies on several other procedures: atoms, MinCorePrio and ExtractNewPreds. The procedure atoms is simply defined as

$$\mathsf{atoms}(\varphi_1 \wedge \ldots \wedge \varphi_n) = \{\varphi_1, \ldots, \varphi_n\} .$$

It takes a conjunction φ and returns the set of its conjuncts.

The procedure MinCorePrio takes three arguments. The first one is a conjunction φ, the second one is an arbitrary formula φ' and the third one is a list L of formulas. As precondition, φ and φ' must be inconsistent. The procedure computes a minimal core of the conjunction φ that is inconsistent with the second argument φ'. There is usually more than one core that can be returned. This choice can be controlled by means of L, the third argument. MinCorePrio gives priority to the set of formulas in L to appear in the resulting minimal core, as illustrated by Algorithm 3. The list L is sorted in ascending order according the priority of its elements. The algorithm proceeds by eliminating irrelevant predicates (conjuncts) of lesser priority (front) first. A predicate is irrelevant if its removal does not have an impact on the inconsistency of the new conjunction with φ'. The lowest priority is given to basic predicates in φ by storing them in the front of the list L (line 6). The consistency test at line 12 of the algorithm is carried out by calling a theorem prover.

Finally, procedure ExtractNewPreds implements a heuristic for predicate inference. It takes a conjunction as argument and returns a list of predicates sorted in ascending order of their likely importance to the convergence of the main CEGAR loop. We will describe this procedure in more details later in this section.

Back to the main procedure Refine, we see that it first computes the weakest precondition (pre) for each of the two traces taken as parameters (lines 4 and 5) to obtain formulas ψ_E and ψ_F. It then applies ExtractNewPreds to augment ψ_E and ψ_F with new facts induced by the two formulas (lines 6, 7, 9, 10). Finally, the minimal unsatisfiabile cores of ψ_E and of ψ_F are computed (lines 8 and 11)

Algorithm 2: Refine

```
Input: two traces $\pi_E$ and $\pi_F$
Output: set of predicates $P$
Var $P$ (initially empty), $S_E$, $S_F$: set of formulas;
Var $P_{new}$: list of formulas;
Var $\psi_E$, $\psi_F$: formula;
$\psi_E := \mathsf{pre}(\pi_E, \mathsf{bad})$;
$\psi_F := \mathsf{pre}(\pi_F, \mathsf{final})$;
$P_{new} := \mathsf{ExtractNewPreds}(\psi_E)$;
$\psi_E := \psi_E \wedge (\bigwedge_{p \in P_{new}} p)$;
$\psi_E := \mathsf{MinCorePrio}(\psi_E, \psi_F, P_{new})$;
$P_{new} := \mathsf{ExtractNewPreds}(\psi_F)$;
$\psi_F := \psi_F \wedge (\bigwedge_{p \in P_{new}} p)$;
$\psi_F := \mathsf{MinCorePrio}(\psi_F, \psi_E, P_{new})$;
$P := P \cup \mathsf{atoms}(\psi_E) \cup \mathsf{atoms}(\psi_F)$ ;
Let $\pi_E = \tau_1; \ldots; \tau_i$;
Let $\pi_F = \tau'_1; \ldots; \tau'_j$;
Let $S_E = \bigcup_{k=1}^{i} \{\varphi_k\}$ s.t. $\varphi_k \equiv \mathsf{pre}(\tau_k; \ldots; \tau_i, \mathsf{bad})$;
Let $S_F = \bigcup_{k=1}^{j} \{\varphi'_k\}$ s.t. $\varphi'_k \equiv \mathsf{pre}(\tau'_k; \ldots; \tau'_j, \mathsf{final})$;
foreach $k$ in range $\{1, \ldots, i-1\}$ do
    $P_{new} := \mathsf{ExtractNewPreds}(\varphi_{k+1})$;
    $\varphi_{k+1} := \varphi_{k+1} \wedge (\bigwedge_{p \in P_{new}} p)$;
    $\psi_E := \mathsf{MinCorePrio}(\varphi_{k+1}[X'/X], \tau_k \wedge \neg\psi_E, P_{new}[X'/X])$;
    $P := P \cup \mathsf{atoms}(\psi_E[X/X'])$ ;
foreach $k$ in range $\{1, \ldots, j-1\}$ do
    $P_{new} := \mathsf{ExtractNewPreds}(\varphi'_{k+1})$;
    $\varphi'_{k+1} := \varphi'_{k+1} \wedge (\bigwedge_{p \in P_{new}} p)$;
    $\psi_F := \mathsf{MinCorePrio}(\varphi'_{k+1}[X'/X], \tau'_k \wedge \neg\psi_F, P_{new}[X'/X])$;
    $P := P \cup \mathsf{atoms}(\psi_F[X/X'])$ ;
return $P$;
```

and conjuncts appearing in either of them are added to the set of predicates (line 12).

In the next phase of the algorithm, the two formulas ψ_E and ψ_F are used to guide the inference of new predicates from states (φ_k's and φ'_k's) belonging to the error trace π_E (first loop, lines 17–21) and to the safe one π_F (second loop, lines 22–26). Along each trace and for each triple of pre-state ψ, transition τ and post-state φ, we want to compute the minimal core φ_m of φ augmented with facts inferred via ExtractNewPreds such that $\mathsf{pre}(\tau, \varphi_m) \Rightarrow \psi$, i.e., $\varphi_m[X'/X] \wedge \tau \wedge \neg\psi \equiv \mathsf{false}$. This amounts to computing the minimal core of $\varphi_m[X'/X]$ with respect to $\tau \wedge \neg\psi$, as performed in lines 20 and 25 of the algorithm.

The procedure ExtractNewPreds is applied to the states of π_E and π_F, i.e., the φ_k's and φ'_k's of each trace. These states are obtained via a backward analysis of π_E and π_F during the initial phase of the algorithm (lines 4 and 5). As mentioned previously, the operator pre (also wp in our case) is limited in inferring

Algorithm 3: MinCorePrio

```
Input: $\varphi$ a conjunction of formulas, $\varphi'$ a formula, $L$ a list of formulas
Output: a conjunction of formulas
Var $\varphi''$: formula;
Var $S$: set of formulas;
Var $L, L'$: list of formulas;
$S := \mathsf{atoms}(\varphi)$;
$L' := L$;
add elements of $S$ to $L'$ in the front;
add elements of $L$ to $S$;
foreach formula $\varphi_L \in L'$ do
    if $S - \{\varphi_L\} = \emptyset$ then
        return $\varphi_L$;
    $\varphi'' := \bigwedge \varphi \mid \varphi \in S - \{\varphi_L\}$;
    if $\varphi'' \wedge \varphi' \equiv \mathsf{false}$ then
        $S := S - \{\varphi_L\}$;
$\varphi'' := \bigwedge \varphi \mid \varphi \in S$;
return $\varphi''$;
```

relevant predicates, as it fails to generalize. Therefore, procedure MinCorePrio biases the computation of the minimal core by giving priority to predicates found via ExtractNewPreds, which are more likely to be general.

4.3 Predicate inference

The procedure ExtractNewPreds plays a key role. It is based on a system of inference rules in the spirit of [22], where an interpolation procedure [18] is used to find predicates, followed by the application of a system of inference rules to deduce range predicates. In [22], the interpolant provides a concise description of the cause of infeasibility of traces, thus the base formula is already minimal. However, the application of the inference rules may introduce redundancies. In our case, MinCorePrio is applied after inferring the new facts, hence, it prevents the inundation of the system with irrelevant predicates. This is not just an optimization: during our experiments, this step has often made the difference between termination and divergence. The system of inference rules that we are using is given in Figure 4.

Predicate inference system. Divergence of the refinement process is often caused by predicates over variables that are increasing or decreasing (counters). This leads to the generation of sequences of constants when loops are effectively unfolded. Another cause of divergence are arrays with counter variables in their index expressions. A simple solution, advocated by [21], is to (initially) discard such predicates.

$$\frac{c_1.e + e_1 \geq 0\ ,\ -c_2.e + e_2 \geq 0}{\begin{array}{c} c_2.e_1 - c_1.e_2 \geq 0 \\ (c_1, c_2 > 0) \end{array}}\ (\text{ELIM}) \qquad \frac{x - e \geq 0\ ,\ -x + e \geq 0}{x = e}\ (\text{EQ})$$

$$\frac{\varphi(x)\ ,\ x = e}{\varphi(e)}\ (\text{SUB}) \qquad \frac{\varphi(i),\ \neg\varphi(j)\ (i < j)}{\exists x \in \{i, \ldots, j\}.\ \varphi(x),\ \exists x \in \{i, \ldots, j\}.\ \neg\varphi(x)}\ (\text{EXIST})$$

$$\frac{\exists x \in \{i, \ldots, j\}.\ \varphi(x),\ j \leq k}{\exists x \in \{i, \ldots, k\}.\ \varphi(x)}\ (\text{EXT\_R}) \qquad \frac{\exists x \in \{i, \ldots, j\}.\ \varphi(x),\ k \leq i}{\exists x \in \{k, \ldots, j\}.\ \varphi(x)}\ (\text{EXT\_L})$$

$$\frac{\varphi(i)}{\forall x \in \{i\}.\ \varphi(x)}\ (\text{UNIV}) \qquad \frac{\forall x \in \{j, \ldots, i\}.\ \varphi(x)\ ,\ \forall x \in \{i+1, \ldots, k\}.\ \varphi(x)}{\forall x \in \{j, \ldots, k\}.\ \varphi(x)}\ (\text{LINK})$$

i and j are integer variables appearing in a linear index expression in φ $(\neg\varphi)$.

Fig. 4. Rules for predicate inference

The aim of he system of rules of Figure 4 is to eliminate likely diverging sequences of predicates whenever possible by inferring new predicates that are more general. Among the symbols used in the system, e refers to linear terms, x is a variable and φ is a formula. The rule ELIM linearly combines two constraints to eliminate common variables. Rule EQ infers equality constraints, which might be used by rule SUB to substitute occurrences of variables with equal terms. The rule UNIV builds a quantified formula and LINK bridges the intervals of two quantified formulas. Finally, the rule EXIST produces two existentially quantified formulas and the rules EXT_R and EXT_L extend the interval of an existentially quantified formula from the right and the left, respectively.

The procedure ExtractNewPreds (Algorithm 4) applies the rules of the inference system to the conjuncts of the formula given as argument and returns a list of predicates sorted in ascending order of priority. A predicate p_1 has higher priority than predicate p_2 if p_1 is produced by a rule where p_2 appears as one of its antecedents. The procedure starts with the list of basic predicates that are extracted from the formula given as argument. These predicates have the lowest priority. It then keeps applying the rules to predicates in the list until saturation, i.e., until no new predicates are produced. The code fragment from line 13 to 17 stores the new predicates according to their priority, i.e., in a position of the list that is beyond the positions of the associated antecedents.

The algorithm terminates, as the two rules ELIM and EQ are only applied to basic predicates (condition at line 12). Thus, they will be called a finite number of times generating a finite number of linear constraints. All other rules will generate a finite number of predicates, as they all depend on linear constraints. We furthermore do not consider nested array expressions. The order in which the rules are applied does not matter.

Illustration Let us illustrate the application of procedure Refine to program copy_2 of Figure 1(b). We call Refine with the error trace $\langle \ell_0, \ell_1, \ell_2, \ell_1, \ell_2, \ell_E \rangle$ and the safe trace $\langle \ell_0, \ell_1, \ell_2, \ell_1, \ell_F \rangle$, which both enter the loop in program copy_2

Algorithm 4: ExtractNewPreds

```
   Input: formula $\varphi$
   Output: list of formulas
1  Var $S, S_b$: set of formulas;
2  Var $L, L'$: list of formulas;
3  Var $R$: list of inference rules;
4  $S_b := \mathsf{atoms}(\varphi)$;
5  insert elements of $S_b$ in $L'$;
6  $R := \{\mathsf{ELIM}, \mathsf{EQ}, \mathsf{UNIV}, \mathsf{SUB}, \mathsf{LINK}, \mathsf{EXIST}, \mathsf{EXT\_L}, \mathsf{EXT\_R}\}$;
7  repeat
8      $L := L'$;
9      foreach rule $r \in R$ do
10         Let $k$ be the number of premises of $r$;
11         foreach tuple $t \in L^k$ do
12             if $(r \notin \{\mathsf{ELIM}, \mathsf{EQ}\}) \vee (\exists i \in \{1, \ldots, k\}\ s.t.\ t_i \notin S_b)$ then
13                 $S := r(t)$;
14                 Let $pos = \max\{pos_j \mid j \in \{1, \ldots, k\} \wedge L[pos_j] = t_j\}$;
15                 foreach predicate $p \in S$ do
16                     if $p \notin L'$ then
17                         insert $p$ after position $pos$ in $L'$;
18 until $L = L'$;
19 return $L$;
```

once. The analysis of these two traces is illustrated in Figure 5. The upper table shows results for the error trace and the lower one for the safe trace. In both tables, the first column contains the suffix of the trace that is analyzed backwards using the weakest precondition. The result is shown in the second column. Finally, the third column shows the new predicates that are inferred using the information from the second column. The superscript associated with each predicate is its priority. At the initial location, which corresponds to the second line in both tables, the predicate $\forall x \in \{0, \ldots, a\_l\}.\ b[x] \neq 0$ is the one with the highest priority for the error trace. It is inferred via the application of the rule UNIV followed by SUB.

For the safe trace, we have two predicates of highest priority, namely $\exists x \in \{0, \ldots, a\_l\}.\ b[x] \neq 0$ and $\exists x \in \{0, \ldots, a\_l\}.\ b[x] = 0$. They are both generated by applying rules EXIST and EXT_R successively. The refinement procedure selects the second predicate as it is the one which separates the two initial states. The selected predicates are underlined in both tables. Going one step backward from the initial location ℓ_0 to location ℓ_1 in both traces, the selected predicates are $\forall x \in \{i, \ldots, a\_l\}.\ b[x] \neq 0$ and $\exists x \in \{i, \ldots, a\_l\}.\ b[x] = 0$ for the error and safe trace, respectively. These are the predicates on which the ones selected at the initial location ℓ_0 depend. One can assert that these two predicates are backwards invariants with respect to the cycle $\langle \ell_1, \ell_2, \ell_1 \rangle$. They thus cover an infinite number of traces.

Error trace	WP	New predicates
$\ell_1, \ell_2, \ell_1, \ell_2, \ell_E$	$i+1 \geq a\_l, b[i+1] \neq 0, i < a\_l$ $b[i] \neq 0$	$a\_l = i+1^{\langle 1 \rangle}, \forall x \in [i, i+1].\ b[x] \neq 0^{\langle 1 \rangle}$ $\underline{\forall x \in [i, a\_l].\ b[x] \neq 0^{\langle 2 \rangle}}$
$\ell_0, \ell_1, \ell_2, \ell_1, \ell_2, \ell_E$	$1 \geq a\_l, b[1] \neq 0, 0 < a\_l$ $b[0] \neq 0$	$a\_l = 1^{\langle 1 \rangle}, \forall x \in [0, 1].\ b[x] \neq 0^{\langle 1 \rangle}$ $\underline{\forall x \in [0, a\_l].\ b[x] \neq 0^{\langle 2 \rangle}}$

Safe trace	WP	New predicates
$\ell_1, \ell_2, \ell_1, \ell_F$	$b[i] \neq 0, i+1 \leq a\_l, b[i+1] = 0$	$\exists x \in [i, i+1].\ b[x] \neq 0^{\langle 1 \rangle}, \exists x \in [i, i+1].\ b[x] = 0^{\langle 1 \rangle}$ $\exists x \in [i, a\_l].\ b[x] \neq 0^{\langle 2 \rangle}, \underline{\exists x \in [i, a\_l].\ b[x] = 0^{\langle 2 \rangle}}$
$\ell_0, \ell_1, \ell_2, \ell_1, \ell_F$	$b[0] \neq 0, 1 \leq a\_l, b[1] = 0$	$\exists x \in [0, 1].\ b[x] \neq 0^{\langle 1 \rangle}, \exists x \in [0, 1].\ b[x] = 0^{\langle 1 \rangle}$ $\exists x \in [0, a\_l].\ b[x] \neq 0^{\langle 2 \rangle}, \underline{\exists x \in [0, a\_l].\ b[x] = 0^{\langle 2 \rangle}}$

Fig. 5. Illustration of algorithm Refine on program copy_2. The underlined predicates are selected by the refinement process. The superscript is the priority of each predicate.

5 Experimental results

Implementation We have implemented our precondition inference technique in the P-Gen[4] tool. P-Gen takes as input a C program containing a procedure annotated with an assertion to be verified. As output, it returns a formula that represents the set of pre-states from which the specified assertion holds for any execution.

Experiments We performed experiments using a desktop computer with 3.7 GB of RAM and a quad-core processor with 2.83 GHz, running Linux. P-Gen uses several theorem provers to compute the abstraction and analyze counterexamples. We have initially used Yices [15] and Simplify [14], but observed limitations when handling quantified formulas. These limitations often lead to the divergence of CEGAR, as the refinement procedure picks up a set of quantifier-free predicates instead of a quantified predicate, and thus fails to generalize. We have subsequently integrated Z3 [13] and used it running as a standalone process communicating with P-Gen through pipes. The Z3 theorem prover was able to decide many queries that were not handled by the two other theorem provers.

The results of our experiments are summarized in Table 1. The column "Precondition" shows the type of precondition inferred ("Q" stands for quantified and "S" stands for simple, i.e., quantifier-free). The column "Iter.U." ("Iter.S.") gives the number of iterations performed by CEGAR to compute the set of unsafe (safe) states and column "Pred.U." ("Pred.S.") gives the number of predicates inferred to abstract the set of unsafe (safe) states. Our tool is based on lazy abstraction [19], we therefore provide the average number of predicates per location instead of the total number of predicates. This number is an indicator for memory consumption, as predicates are encoding program states. Finally,

[4] http://www.cs.ox.ac.uk/people/nassim.seghir/pgen-web-page

Program	Precond.	Iter.U.	Pred.U.	Iter.S.	Pred.S.	Alt.	Time (s)
memcmp	Q + S	5	5	8	3	2	23.64
strcat	Q	4	4	4	3	2	0.77
memchr	Q + S	5	4	8	3	2	77.30
strlen	Q	4	4	4	3	2	0.80
memcpy	S	3	2	3	2	1	0.17
memmove	S	5	4	9	2	2	0.91
strchr	Q	5	7	7	4	2	1.92
r_strcat	Q	5	2	3	2	2	9.02
strcspn	Q	5	6	7	4	2	7.30
strspn	Q	5	6	7	4	2	7.05
my_strcmp	Q	5	7	7	4	2	2.70
my_strcpy	Q	4	4	4	3	2	1.41
AllNotNull	Q + S	6	5	5	3	2	2.07
perfc_copy_info	S	7	2	13	2	2	1.94
bitmap_shift_right	S	15	13	26	5	2	41.60
mvswap	S	7	5	8	4	3	0.70
BZ2_hbAssignCodes	S	8	4	6	4	2	0.73

Table 1. Experimental results for routines taken from the C string library (upper part) and from real-world programs (lower part)

column "Alt." refers to the number of alternations between the two abstractions (i.e., the number of times the procedure switches between the abstraction of unsafe states and the abstraction of safe states).

The upper part of the table relates to implementations of routines from the C string library[5]. The last two procedures, my_strcmp and my_strcpy, are modified versions of the original strcmp and strcpy. In the lower part of the table, we have AllNotNull, which was used by Cousot et al. as illustration [12]. In their paper, they propose the precondition $\forall i.\ 0 \leq i < A.length \Rightarrow A[i] \neq null$. Surprisingly, our tool infers a weaker precondition, as it considers the case that the length of A can be zero. We obtain $(\forall i.\ 0 \leq i < A.length \Rightarrow A[i] \neq null) \vee A.length \leq 0$. We do not know if this case is missing in [12] or if the length of A was assumed to be strictly positive. The procedures mvswap and BZ2_hbAssignCodes are taken from the source code of the compression program Bzip2[6]. The procedures perfc_copy_info and bitmap_shift_right belong to the Xen Hypervisor[7].

Although the refinement heuristic performs well in most of the cases, sometimes it does not select adequate predicates in early stages of the iterative process (as for memchr, memcmp and bitmap_shift_right), resulting in high time consumption. However, due to the precision of the precondition, this cost can often be amortised when checking code that uses the functions.

[5] http://en.wikibooks.org/wiki/C_Programming/Strings.
[6] http://www.bzip.org/
[7] http://www.xen.org/products/xenhyp.html

6 Related Work

The work presented in this paper is linked to several topics: predicate abstraction, invariant generation, counterexample-guided refinement, modular verification and precondition inference. Along these axes, we elaborate on some related work from the literature.

The combination of predicate abstraction [16] with counterexample-guided abstraction refinement [9] has been pioneered by the SLAM [1] tool at Microsoft, and has subsequently been implemented in many other tools, including BLAST [19], MAGIC [7], ARMC [24], F-SOFT [20] and SATABS [8]. The DASH [2] and SYNERGY [17] algorithms are variants of this approach where program testing is used to refine abstractions. All these methods and tools check the validity of a given assertion. In contrast, we use CEGAR to compute a precondition under which the assertion is valid. Our technique represents a general scheme for extending the previously mentioned tools to infer preconditions, as it uses ingredients that are common to all of them.

Moy has proposed a technique to infer preconditions [23] using a combination of state-of-the-art techniques such as abstract interpretation, weakest preconditions and quantifier elimination. While his technique is stronger than many existing ones, it is unable to infer quantified preconditions. Our technique is able to infer universally- as well as existentially-quantified preconditions. Blanc and Kroening proposed a technique to optimize the simulation of SystemC code [3]. It consists of inferring conditions that are sufficient for the commutativity of pairs of processes. Like our algorithm, theirs also is based on CEGAR. However, they have no guarantee that the inferred precondition is exact. Our method does provide this guarantee.

Taghdiri proposed an approach for generating approximations of relations (over pre- and post-states) induced by functions [27]. A function specification (relation) is computed with respect to a context that includes the property to be verified, thus the specification may lack generality. Moreover, the number of times loops are unrolled is bounded, making the approach unsuitable for proving the absence of bugs. Our technique over-approximates the set of all behaviors. Thus, the preconditions computed by our method guarantee safety.

Sankaranarayanan et al. presented a technique that combines test and machine learning to infer likely data preconditions over a set of predicates [26]. In many cases, they were able to learn preconditions that ensure safe executions. However, as their technique is based on testing, it can only suggest preconditions, but does not guarantee their soundness.

In the context of abstract interpretation, Cousot et al. formulated the contract inference problem for intermittent assertions precisely [12]. The precondition extracted by their method does not exclude safe runs even when a non-deterministic choice could lead to bad ones. Our treatment of non-determinism is different, as we report a warning to the user. The method described in [12] as well as [4] and [25] rely on some predefined abstract domains. In our approach, the precision of the abstraction is automatically tuned as required by means of CEGAR. We can also enhance the refinement process by introducing new inference rules, without having to implement new transformers.

Calcagno et al. presented a technique based on Bi-Abduction to infer pre- and post-specifications of the heap [6]. One of the major advantages of their approach is scalability. Similar to their approach, we intend to complete the picture by integrating our method into an inter-procedural reasoning framework. Although we can deal with pointers, the properties handled by their technique are out of the scope for our tool as we do not have a theory to reason about heap properties. On the other side, contrary to our approach, the preconditions they compute are not exact, meaning that they might have to be refined. In the context of termination, Bozga et al. [5], proposed a method to generate exact preconditions for a restricted class of programs. Other techniques for inferring preconditions for termination are applicable to a larger set of programs [10], however they only generate sufficient preconditions.

7 Conclusion

We have presented a new method for precondition inference based on counterexample-guided abstraction refinement. Given a procedure containing an assertion, our method infers a formula that is sufficient for the validity of the specified assertion and exclusively refers to state variables visible at the entry point of the procedure. The inferred precondition is independent of the context, making it reusable. The computed precondition is also necessary for the validity of the assertion, i.e., it does not exclude any safe runs of the procedure. Hence, we avoid false alarms, as the violation of the precondition corresponds to a real error.

Our technique is based on ingredients commonly used in CEGAR-based assertion checkers, and it thus represents a general scheme for extending other verification tools to infer preconditions. In addition to that, we believe that we can take advantage of components from other tools, such as advanced refinement mechanisms, which can be integrated seamlessly into our algorithm. Preliminary experimental results are encouraging, as we are able to generate exact preconditions (quantified as well as quantifier-free) for string- and array-manipulating programs.

References

1. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
2. N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA*, pages 3–14, 2008.
3. N. Blanc and D. Kroening. Race analysis for SystemC using model checking. In *Proceedings of ICCAD 2008*, pages 356–363. IEEE, 2008.
4. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI*, pages 46–55, 1993.
5. M. Bozga, R. Iosif, and F. Konecný. Deciding conditional termination. In *TACAS*, pages 252–266, 2012.

6. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.
7. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395, 2003.
8. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.
9. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
10. B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *CAV*, pages 328–340, 2008.
11. B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *CAV*, pages 415–418, 2006.
12. P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *VMCAI*, pages 150–168, 2011.
13. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
14. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Lab., 2003.
15. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, pages 81–94, 2006.
16. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
17. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127, 2006.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
20. F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-SOFT. In *ICCD*, pages 297–308, 2005.
21. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.
22. R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.
23. Y. Moy. Sufficient preconditions for modular assertion checking. In *VMCAI*, pages 188–202, 2008.
24. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, volume 4354 of *LNCS*, pages 245–259, 2007.
25. X. Rival. Understanding the origin of alarms in Astrée. In *SAS*, pages 303–319, 2005.
26. S. Sankaranarayanan, S. Chaudhuri, F. Ivancic, and A. Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *ISSTA*, pages 295–306, 2008.
27. M. Taghdiri. Inferring specifications to detect errors in code. In *ASE*, pages 144–153, 2004.