Automatic Inference of Necessary Preconditions

Patrick Cousot Radhia Cousot Manuel Fähndrich Francesco Logozzo
NYU, ENS, CNRS, INRIA CNRS, ENS, INRIA Microsoft Research
pcousot@cims.nyu.edu rcousot@ens.fr {maf,logozzo}@microsoft.com

Abstract. We consider the problem of *automatic* precondition inference. We argue that the common notion of *sufficient* precondition inference (*i.e.*, under which precondition is the program correct?) imposes too large a burden on callers, and hence it is unfit for automatic program analysis. Therefore, we define the problem of *necessary* precondition inference (*i.e.*, under which precondition, if violated, will the program *always* be incorrect?). We designed and implemented several new abstract interpretation-based analyses to infer atomic, disjunctive, universally and existentially quantified necessary preconditions.
We experimentally validated the analyses on large scale industrial code. For unannotated code, the inference algorithms find necessary preconditions for almost 64% of methods which contained warnings. In 27% of these cases the inferred preconditions were also *sufficient*, meaning all warnings within the method body disappeared. For annotated code, the inference algorithms find necessary preconditions for over 68% of methods with warnings. In almost 50% of these cases the preconditions were also sufficient. Overall, the precision improvement obtained by precondition inference (counted as the additional number of methods with no warnings) ranged between 9% and 21%.

1 Introduction

Design by Contract [28] is a programming methodology which systematically requires the programmer to provide the preconditions, postconditions and object invariants (collectively called contracts) at design time. Contracts allow automatic generation of documentation, amplify the testing process, and naturally enable assume/guarantee reasoning for divide and conquer static program analysis and verification. In the real world, relatively few methods have contracts that are sufficient to prove the method correct. Typically, the precondition of a method is weaker than necessary, resulting in unproven assertions within the method, but making it easier to prove the precondition at call-sites. Inference has been advocated as the holy grail to solve this problem.

In this paper we focus on the problem of computing *necessary preconditions* which are *inevitable checks* from within the method that are hoisted to the method entry. This should be contrasted to *sufficient preconditions* which guarantee the absence of possible assertion violations inside the method but may rule out good runs. Programmers will object to the inference of too strong preconditions as they introduce more false warnings at call sites.

Contribution. The basis for this work is [11] which introduced the theoretical framework formalizing the notion of necessary precondition inference. We extend this theoretical framework by introducing the inter-method analysis, by refining the intra-method analyses combining them with `Clousot` [16], by identifying under which circumstances the generated precondition is also sufficient, by showing how one can infer existential preconditions, by introducing a simplification step for scalability, by adding the provenance relation, and by implementing and validating the approach on realistic code bases.

2 Semantics

The semantics of a given program `P` (*e.g.* a method) is a non-empty set $\mathcal{S}$ of runs modeled as finite or infinite execution traces over states Σ. $\mathcal{S}$ can be partitioned into disjoint subsets $\mathcal{S} = \mathcal{E} \cup \mathcal{T} \cup \mathcal{I}$ where the traces in $\mathcal{E}$ are finite bad runs terminating in an error state, the traces in $\mathcal{T}$ are finite good runs terminating in a correct state, and the infinite traces in $\mathcal{I}$, which correspond to non-termination. If X is a set of traces and $s \in \Sigma$, we write $X(s)$ for the set of traces in X starting from state s.

Assertions are either induced by the language semantics (*e.g.*, null-pointer dereference, division by zero, array out of bounds, ...) or they are annotations in the source text (programmer-provided assertions, preconditions, and postconditions). Boolean expressions are side effect free and they are always well-defined when evaluated with shortcut semantics for conjunctions and disjunctions. The set $\mathbb{A}$ denotes the set of the potential failures of `P`. $\mathbb{A}$ contains pairs $\langle \mathtt{c}, \mathtt{b} \rangle$, where `b` is an assertion at the program point `c`. In general, there may be more than one assertion per program point. The bad runs $\mathcal{E} \triangleq \{\sigma s' \mid \exists \langle \mathtt{c}, \mathtt{b} \rangle \in \mathbb{A} : \pi s' = \mathtt{c} \wedge \neg [\![\mathtt{b}]\!] s'\}$ are all traces $\sigma s'$ ending in a state $s' \in \Sigma$ at a control point $\pi s' = \mathtt{c}$ where the evaluation of a language or programmer assertion `b` fails, that is $[\![\mathtt{b}]\!] s'$ is false.

Given a formula $C \Rightarrow S$, we say that C is a sufficient condition for S and S is a necessary condition for C. A *sufficient condition* for a statement S is a condition that, *if* satisfied, ensures S's correctness. A *necessary condition* for a statement S *must* be satisfied for the statement to be true.

The first step in our precondition inference algorithm is the collection of all failure points $\langle \mathtt{c}, \mathtt{b} \rangle$ from which we will then try to derive *necessary* preconditions. In practice, the candidate assertions in $\mathbb{A}$ are those assertions which cannot be statically proven by `cccheck` [16](or similar tools). We will use the assertions in $\mathbb{A}$ to infer necessary preconditions. This consists in propagating these conditions backwards to the origin of the traces of the semantics $\mathcal{S}$, at the `entry` control point. The inference of termination preconditions is a separate problem [10], so we ignore the non-terminating behaviors $\mathcal{I}$, or equivalently, assume termination *i.e.* $\mathcal{I} = \emptyset$.

3 Sufficient Preconditions

The weakest (liberal) preconditions provide *sufficient* preconditions which guarantee the (partial) correctness, *i.e.*, the absence of errors in the program [2,5,9,22,29,31]:

```
public static void Example(object[] a) {
  Contract.Requires(a != null);
  for (var i = 0; i <= a.Length; i++) {
    a[i] = ...f(a[i])... ;  // (*)
    if (NonDet()) return;
  }
}
```

Fig. 1. The weakest precondition for this code is *false*, which rules out *good* executions. Our technique only excludes *bad* runs, inferring the necessary precondition $\texttt{0} < \texttt{a.Length}$.

$$\forall s \in \Sigma : \mathsf{wlp}(\mathtt{P}, \mathsf{true})(s) \triangleq (\mathcal{E}(s) = \emptyset).$$

The main drawbacks preventing the use of the weakest (liberal) preconditions calculus for automatic precondition inference are: (i) in the presence of loops, there is no algorithm that computes weakest (liberal) precondition $\mathsf{wlp}(\mathtt{P}, \mathsf{true})$, (ii) due to loop over-approximation, the inferred preconditions are sufficient but no longer the weakest, and (iii) the resulting sufficient (liberal) preconditions may be too strong and rule out good runs.

More formally, an *under-approximation* $\overline{P}$ of $\mathsf{wlp}(\mathtt{P}, \mathsf{true})$ on states $s \in \Sigma$ must be computed such that

$$\begin{aligned}
& \forall s \in \Sigma : \overline{P}(s) \Rightarrow \mathsf{wlp}(\mathtt{P}, \mathsf{true})(s) && \wr\text{under-approximation}\wr \\
\Leftrightarrow\quad & \forall s \in \Sigma : \overline{P}(s) \Rightarrow (\mathcal{E}(s) = \emptyset) && \wr\text{def. } \mathsf{wlp}(\mathtt{P}, \mathsf{true})\wr \\
\Leftrightarrow\quad & \forall s \in \Sigma : \overline{P}(s) \Rightarrow (\mathcal{T}(s) \neq \emptyset \vee \mathcal{I}(s) \neq \emptyset) \\
& \qquad \wr\text{since } \mathcal{S}(s) \neq \emptyset \wedge \mathcal{S} = \mathcal{E} \cup \mathcal{T} \cup \mathcal{I} \text{ so } \mathcal{E}(s) = \emptyset \text{ implies } (\mathcal{T}(s) \cup \mathcal{I}(s)) \neq \emptyset.\wr \\
\Leftrightarrow\quad & \forall s \in \Sigma : [\mathcal{I}(s) = \emptyset] \Rightarrow [\overline{P}(s) \Rightarrow (\mathcal{T}(s) \neq \emptyset)] && (1)
\end{aligned}$$

The precondition $\overline{P}$ on states $s \in \Sigma$ is *sufficient* for the absence of definite runtime errors (under the termination hypothesis) but $\neg\overline{P}(s) \wedge (\mathcal{T}(s) \neq \emptyset)$ is possible so if execution stops in states s such that $\neg\overline{P}(s)$ (the sufficient precondition fails) then $\mathcal{T}(s) \neq \emptyset$ implies that other valid executions may be ruled out.

We argue that in general the use of a sufficient precondition is unfair in an automatic static analysis assume/guarantee reasoning setting, as: (i) it rules out correct executions; and (ii) it imposes too strong a proof obligation at call-sites.

Example 1 (Ruling out good runs). Consider the code in Fig. 1, a very simplified version of some pattern we found in `System.dll`. The function `NonDet` is a non-deterministic Boolean function. At runtime, an out-of-bounds array access at line $(*)$ may or may not appear. If the array is empty then the execution will fail while trying to access the first element. Otherwise, the failure will not appear if some of the `NonDet` calls returns `true`.

The weakest (liberal) precondition is *false*, meaning that all the runs, even the good ones, are rejected. To us, this is too strict. We propose a definition where no *good* run is removed, but only *bad* ones. For instance according to our definition, it is correct for our automatic tool to infer the precondition $\texttt{a.Length} > \texttt{0}$ as an empty array will definitely lead to a failure at runtime. □

```
int Sum(int[] xs)
{
  Contract.Requires(xs != null);

  int sum = 0;
  for(var i = 0; i < xs.Length; i++)
    sum += xs[i];

  Assert(sum >= 0);

  return sum;
}
```

Fig. 2. In presence of overflows, the weakest precondition of `Sum` is essentially the method itself. The weakest precondition can be overapproximated (as customary in deductive verification) with a *sufficient* precondition. Automatically inferred sufficient preconditions may require the caller to establish a too strict condition.

Example 2 (Requiring too much from the client). Let us consider the method in Fig. 2, where `int` is a 32-bit integer, and overflows are not an error, but a desired side-effect. `Sum` returns 19 for the input array $\{-2147483639, -2147483628, -10\}$. The weakest precondition of the method `Sum` is essentially the method itself:

$$\left(\overline{\sum}_{0 \leq j < \mathtt{xs.Length}} \mathtt{xs[i]}\right) \geq 0. \tag{2}$$

It is a second order formula as $\overline{\sum}$, the sum on two's complement is defined inductively. The automatic inference of (2) is tough, out of reach of the current state-of-the-art tools and inference techniques. One can imagine tools inferring weaker loop invariants, originating in stronger sufficient preconditions. Two possible sufficient preconditions for the method are

$$\forall j \in [0, \mathtt{xs.Length}).\ 0 \leq \mathtt{xs}[j] < \mathtt{MaxInt}/\mathtt{xs.Length} \tag{3}$$

$$\mathtt{xs.Length} = \mathtt{3} \land \mathtt{xs}[0] + \mathtt{xs}[1] = 0 \land \mathtt{xs}[2] \geq 0. \tag{4}$$

as they satisfy the corresponding Hoare triples $\{(3)\}\ \mathtt{C}\ \{Q\}$, $\{(4)\}\ \mathtt{C}\ \{Q\}$ and $\{(3) \lor (4)\}\ \mathtt{C}\ \{Q\}$. However, it is unfair to use one of them for an automatic modular assume/guarantee reasoning. For example, the input array above satisfies neither (3) nor (4). So, a tool inferring a sufficient but not necessary precondition will report a precondition violation for a caller with such an actual parameter. □

A sufficient precondition may impose too large a burden on callers, thereby making the precondition appear wrong to the user. In an early attempt at precondition inference, we were inferring sufficient but not necessary preconditions. Our users (professional programmers with no background in formal methods) filed several bug reports, marking such preconditions as *"wrong"* suggestions from `cccheck`.

4 Necessary Preconditions

We advocate the use of necessary preconditions, *i.e.*, preconditions which, once violated, definitely lead to an error later in the program execution. Such a *necessary* precondition $\underline{P}$ on states $s \in \Sigma$ is

		$\mathcal{T}(s)$	
	$\overline{P}(s)$	$\emptyset$	$\neq\emptyset$
$\mathcal{E}(s)$	$\emptyset$	true	true
	$\neq\emptyset$	false	false

$\overline{P}(s) = \mathsf{wlp}(\mathsf{P}, \mathsf{true})(s) \triangleq (\mathcal{E}(s) = \emptyset)$

		$\mathcal{T}(s)$	
	$\underline{P}(s)$	$\emptyset$	$\neq\emptyset$
$\mathcal{E}(s)$	$\emptyset$	true	true
	$\neq\emptyset$	false	true

$\underline{P}(s) = (\mathcal{T}(s) \neq \emptyset \vee \mathcal{E}(s) = \emptyset)$

Fig. 3. (a) Weakest *sufficient* precondition (b) Strongest *necessary* precondition

$$
\begin{array}{lll}
 & \forall s \in \Sigma : [\mathcal{I}(s) = \emptyset] \Rightarrow [(\mathcal{T}(s) \neq \emptyset) \Rightarrow \underline{P}(s)] & \wr\text{inverse of (1)}\int \\
\Leftrightarrow & \forall s \in \Sigma : [\mathcal{I}(s) = \emptyset] \Rightarrow [\neg\underline{P}(s) \Rightarrow (\mathcal{T}(s) = \emptyset)] & \wr\text{contraposition}\int \\
\Leftrightarrow & \forall s \in \Sigma : [\mathcal{I}(s) = \emptyset] \Rightarrow [\neg\underline{P}(s) \Rightarrow (\mathcal{T}(s) = \emptyset \wedge \mathcal{E}(s) \neq \emptyset)] & \\
 & \wr\text{since } \mathcal{S}(s) \neq \emptyset,\ \mathcal{S} = \mathcal{E} \cup \mathcal{T} \cup \mathcal{I},\ \mathcal{I}(s) = \emptyset, \text{ and } \mathcal{T}(s) = \emptyset \text{ imply } \mathcal{E}(s) \neq \emptyset.\int &
\end{array}
$$

If the necessary precondition $\underline{P}(s)$ does not hold then an execution from s either diverges or else it definitely terminates in an error (since $\mathcal{T}(s) = \emptyset$ so there is no possible finite correct execution). Setting apart infinite traces (*i.e.* $\mathcal{I}(s) = \emptyset$), Fig. 3 shows that the difference is only when $\mathcal{E}(s) \neq \emptyset \wedge \mathcal{T}(s) \neq \emptyset$. Whereas the sufficient precondition rules out all correct executions (since $\overline{P}(s) = \mathsf{false}$) the necessary precondition allows all of them (since $\underline{P}(s) = \mathsf{true}$), but maybe including erroneous ones.

5 Intra-Procedural Precondition Inference

We briefly recall and illustrate three of the four algorithms introduced in [11] to infer under-approximations of necessary preconditions, that we have implemented in `Clousot/cccheck`, the static analyzer of CodeContracts. These fully automatic static analyses effectively compute preconditions with concretization $P^\flat$ such that

$$\forall s \in \Sigma : P^\flat(s) \Rightarrow (\mathcal{T}(s) = \emptyset \wedge \mathcal{E}(s) \neq \emptyset) \ .$$

These preconditions $P^\flat$ are therefore sufficient to guarantee the presence of definite errors (*i.e.* runtime error, programmer assertion or post condition failures) or non-termination. So $P^\flat$ is an under-approximation of the error semantics $\mathcal{T}(s) = \emptyset \wedge \mathcal{E}(s) \neq \emptyset$ while its negation $\neg P^\flat$ is weaker than the strongest necessary precondition

$$\forall s \in \Sigma : (\mathcal{T}(s) \neq \emptyset \vee \mathcal{E}(s) = \emptyset) \Rightarrow \neg P^\flat(s) \ .$$

The inferred preconditions are therefore *necessary* in that they do not guarantee (in general) the correctness of the method, but if not established they certainly imply its failure. As shown by the experiments, these necessary preconditions may also be sufficient to prove the correctness of the assertion they originated from.

```
void Partition(int[] array, int left, int right, int pivotIndex) {
 1:  var pivotValue = array[pivotIndex];
 2:  swap(ref array[pivotIndex], ref array[right]);
 3:  var storeIndex = left;
 4:  for (var i = left; i < right; i++)
 5:  {
 6:    if (array[i] < pivotValue) {
 7:      swap(ref array[i], ref array[storeIndex]);
 8:      storeIndex++; }
 9:  }
10:  swap(ref array[storeIndex], ref array[right]);
}
```

Fig. 4. A partitioning routine as found in QuickSort implementations. Our technique infers necessary preconditions (5) and (6) which turn out to be also sufficient.

5.1 All-Paths Precondition Analysis (APPA)

The all-paths precondition analysis (APPA) of [11, Sect. 7] symbolically hoists assertions $\langle \texttt{c}, \texttt{b}\rangle \in \mathbb{A}$ all the way back to the code/method entry. Three conditions should hold: (i) the value of `b` is the same at `c` and at `entry`; (ii) the value of `b` is checked on all paths from the `entry`; and (iii) the variables in `b` have the correct visibility. In general, the generated precondition will be an atomic formula, containing a disjunction if and only if `b` contains one. From the three conditions above, it follows that if `b` holds at entry then it will also hold in all the method paths, therefore the generated precondition is also sufficient (for `b`).

Example 3 (Atomic preconditions (APPA)). Consider the in-place version of the partitioning phase of Quicksort (Fig. 4). In the example, `cccheck` verifies 20 assertions (null-pointer accesses, lower and upper array bounds) and issues 7 warnings. The warnings are the array dereference and the two array bounds checks at line `1`, two array bounds checks at line `2`, the lower bound check at line `6` and the upper bound check `array`[`storeIndex`] at line `10` (the analysis infers `storeIndex` $\geq$ `0` at line `10`). A necessary precondition has to be generated for those warnings. The `array` dereference at line `1` definitely causes an error if the actual value of `array` is null. Same for the array loads at lines `1` and `2`: if `pivotIndex` and `right` are not in the bounds of `array` then the program will fail for sure. These assertions can be pushed up to the entry point to generate the following *necessary* preconditions:

```
Contract.Requires(array != null && 0 <= pivotIndex);
Contract.Requires(pivotIndex < array.Length);
Contract.Requires(0 <= right && right < array.Length);        (5)    □
```

5.2 Conditional-Path Precondition Analysis (CPPA)

The conditional-path precondition analysis (CPPA) of [11, Sect. 9], hoists more assertions $\langle \texttt{c}, \texttt{b}\rangle \in \mathbb{A}$ to the procedure entry point than APPA by taking into account program paths and tests, and using dual widening to cope with infinite path lengths. The basic abstract predicates $\texttt{b}_p \rightsquigarrow \texttt{b}_a$ mean that when the path

```
public void Combination(string x, int z) {
  Contract.Requires(z >= 0);
  while (z > 0)  { z--; }
  // here Clousot infers  z == 0
  if (z == 0)
    Assert(x != null);
}
```

Fig. 5. A simplified example from `mscorlib` where the information inferred by a forward static analysis is used to generate better preconditions.

condition $\mathtt{b}_p$ holds, execution will definitely be followed by an `assert(b)` and checking $\mathtt{b}_a$ at the beginning of the path is the same as checking this `b` later in the path when reaching the assertion. The partial order is $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a \Mapsto \mathtt{b}'_p \rightsquigarrow \mathtt{b}'_a \triangleq \mathtt{b}'_p \Mapsto \mathtt{b}_p \wedge \mathtt{b}_a \Mapsto \mathtt{b}'_a$ where the abstract implication $\mathtt{b} \Mapsto \mathtt{b}'$ underapproximates the concrete implication $\Rightarrow$: $\mathtt{b} \Mapsto \mathtt{b}'$ implies that $\forall s \in \Sigma : [\![\mathtt{b}]\!]s \Rightarrow [\![\mathtt{b}']\!]s$. In general, the analysis will generate atomic preconditions containing disjunctions. If no loops are encountered in the path between `entry` and `c`, then the generated necessary precondition is also sufficient (for `b`).

Example 4. In the `Combination` procedure of Fig. 5, `Clousot` infers that $\mathtt{z} = \mathtt{0}$ after the loop, so that the precondition $\mathtt{x}! = \mathtt{null}$ is generated. Note that the invariant inferred from `Clousot` is crucial to infer the precondition (no precondition would be inferred otherwise). □

Example 5 (Atomic preconditions (CPPA)). Continuing Ex. 3, we are left with two candidate assertions. In one case the assertion is not checked on every path (line `6`, unreached if `left`$\geq$ `right`). In the other case `storeIndex` may have been modified (line `10`). So the APPA analysis above does not apply. With CPPA, the candidate assertions are propagated backwards, taking into account tests and computing a fixpoint. We can then infer the disjunctive necessary preconditions:

```
Contract.Requires(left < right  || left < array.Length);
Contract.Requires(left >= right || 0 <= left);                (6)
```

Informally, the two preconditions state that whenever `left` $<$ `right` then `left` should be non-negative otherwise `left` $<$ `array`.`Length`.

In this case, the inferred necessary preconditions are also sufficient, as it can be easily checked by instrumenting `Partition` with the inferred preconditions and running `cccheck` again. Please note that the preconditions above are weaker than the usual ones found in the specification of the partition algorithm which require $0 \leq$ `left` and `left` $<$ `right` to ensure functional correctness (*e.g.*, in Hoare's original paper on QuickSort [24]). □

5.3 Quantified Precondition Analysis (QPA)

The APPA and CPPA analyses cannot deal directly with unbounded data structures such as collections and arrays. [11, Sect. 10] uses a forward static analysis based on [12] to synthesize quantified preconditions. This quantified

```
void ReadAndConsume(Message[] msg) {
   Contract.Requires(msg != null);
   for (var i = 0; i < msg.Length; i++) {
     Assert(msg[i] != null);
    // Do something with msg[i], then consume it
     msg[i] = null;
   }
  // Here msg[*] == null
}
```

Fig. 6. To prevent an error, `msg` should not contain any `null` values. Inferring this precondition (7) requires non-trivial reasoning about which elements of the array have been tested and modified.

---

precondition analysis (QPA) can deduce that a subset of the collection elements are: (i) checked in every execution path; and (ii) when checked, they have the same value they had at entry, so as to synthesize a *universally* quantified precondition.

Example 6 (Universally quantified preconditions). The method precondition for the example in Fig. 6 is too weak to prevent a runtime error: if `msg` is not empty and one of its elements is `null` then the program will definitely fail at runtime. The precondition should be quantified over the array elements, so the inference of atomic preconditions above does not help. The inference is non-trivial as the content of the input array is modified inside the loop: The analysis should make sure that the checked array elements are the same as in the pre-state. We infer the necessary universally quantified precondition

```
    Contract.Requires(ForAll(message, msg => msg != null));           (7)
```

Please note that: (i) the precondition encompasses the case of an empty array; and (ii) it is also sufficient. □

Inferred necessary preconditions may not be sufficient, in particular when the condition must be weakened during backward propagation due to control flow or loops.

Example 7 (Necessary but not sufficient preconditions). Consider the code in Fig. 7, a simplified version of a common pattern used in C/C++ programs and .NET framework libraries. Pushing the array index condition backwards produces the precondition

$$\texttt{Contract.Requires}(\texttt{a.Length} > 0),$$

which is not sufficient. If we semantically unroll the loop k times, we can generate increasingly stronger necessary preconditions of the form

$$\texttt{a.Length} > 0 \land \texttt{a}[0] = 3 \lor \texttt{a.Length} > 1 \land \texttt{a}[1] = 3 \lor \ldots,$$

yet none of them are sufficient. In general, the precondition inference requires a fixpoint computation over an infinite domain. The convergence of the computation should be enforced using a widening operator. In the weakest precondition calculus, using a widening can very easily bring to the inference of sufficient preconditions. In necessary precondition inference, the *dual* widening can simply stop the

```
int FirstOccurrence(int[] a) {
  Contract.Requires(a != null);
  var i = 0;
  while (a[i] != 3) { i++; }
  return i;
}
```

Fig. 7. An example where all the inferred atomic preconditions are necessary but not sufficient. A sufficient existentially quantified precondition (8) can be inferred by a forward array content and modification analysis.

iterations after k iterations — A widening over-approximates its arguments, while a dual widening under-approximates them. The desired precondition is existentially quantified:

$$\exists j : j \in [0, \texttt{array.Length}) \wedge \texttt{array}[j] = 3 \tag{8}$$

Such a precondition can be inferred by combining forward array content and array modification analyses. $\square$

6 Scaling Up Thanks to Simplification

The disjunctive precondition $P^\flat$ represented as a set $\mathcal{P}_\mathbb{A}$ of terms in $\mathbb{A}$ may contain redundant preconditions which should be removed for two main pragmatic reasons. First, the more preconditions, the more proof obligations need to be discharged in other methods. Second, more preconditions mean more suggestions to the end-user, who may get irritated if they are redundant (as we experienced with `cccheck`).

We would like to compute a minimal yet equivalent set $\mathcal{P}^m_\mathbb{A}$. The set $\mathcal{P}^m_\mathbb{A}$ should be: (i) a set of generators ($\forall \texttt{p} \in \mathcal{P}_\mathbb{A} : \exists\{\texttt{p}_0 \dots \texttt{p}_n\} \subseteq \mathcal{P}^m_\mathbb{A} : \texttt{p}_0 \wedge \dots \wedge \texttt{p}_n \Rightarrow \texttt{p}$); and (ii) minimal ($\forall \texttt{p} \in \mathcal{P}_\mathbb{A} : \bigwedge\{\texttt{q} \mid \texttt{q} \in \mathcal{P}^m_\mathbb{A} \setminus \{\texttt{p}\}\} \not\Rightarrow \texttt{p}$). Unfortunately, computing a minimal set of generators can be very expensive [15] or even impossible when the inferred invariants are quantified. To see why, let $\texttt{p}_1$ and $\texttt{p}_2$ be two Boolean expressions containing quantified facts over arrays, then $\texttt{p}_1 \Rightarrow \texttt{p}_2$ is not decidable [3]. There exist subsets for which the problem is decidable, *e.g.*, equalities or difference constraints. In general the minimal set of generators is not unique.

In practice, we are not interested in getting the best $\mathcal{P}^m_\mathbb{A}$, but only a good approximation. The approximation should be such that: (i) obviously-redundant preconditions are removed; and (ii) it is fast, in that only an infinitesimal fraction of the analysis time allocated for the procedure is spent on the simplification. In our implementation we use a simple heuristics to simplify the candidate preconditions and get a set $\overline{\mathcal{P}}^m_\mathbb{A} \supseteq \mathcal{P}^m_\mathbb{A}$ such that $\#\overline{\mathcal{P}}^m_\mathbb{A} \leq \#\mathcal{P}_\mathbb{A}$, for some minimal set of generators $\mathcal{P}^m_\mathbb{A}$.

The simplification equations are given in Fig. 8. The rationale is that we want to simplify as many disjunctive preconditions as possible, trivial quantified facts, and difference constraints. The precondition `true` or any disjunct containing it

$\mathsf{simpl} \triangleq \boldsymbol{\lambda}\mathcal{P}\,\bullet$

$\quad \mathtt{true} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\mathtt{true}\}$

$\quad \mathtt{true}||\mathtt{b} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\mathtt{true}||\mathtt{b}\}$ (|| is the non-commutative short-cutting disjunction)

$\quad \mathtt{false} \in \mathcal{P} \quad \rightarrow \quad \{\mathtt{false}\}$

$\quad \mathtt{false}||\mathtt{b} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\mathtt{false}||\mathtt{b}\} \cup \{\mathtt{b}\}$

$\quad \mathtt{b}||t \in \mathcal{P} \wedge t \in \{\mathtt{true}, \mathtt{false}\} \quad \rightarrow \quad \mathcal{P} \setminus \{\mathtt{b}||t\} \cup \{t||\mathtt{b}\}$

$\quad \mathtt{b}_1||\mathtt{b},\ \mathtt{b} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\mathtt{b}_1||\mathtt{b}\}$

$\quad \mathtt{b}_1||\mathtt{b},\ !\mathtt{b}_1||\mathtt{b} \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\mathtt{b}_1||\mathtt{b},\ !\mathtt{b}_1||\mathtt{b}\} \cup \{\mathtt{b}\}$

$\quad \mathtt{b}_1||\mathtt{b},\ \mathtt{b}_2||\mathtt{b} \in \mathcal{P} \wedge s(\mathtt{b}_1) \subseteq s(\mathtt{b}_2) \quad \rightarrow \quad \mathcal{P} \setminus \{\mathtt{b}_2||\mathtt{b}\}$

$\quad \mathtt{f} \triangleq \forall i \in [\mathtt{l}, \mathtt{l}+\mathtt{1}) : \mathtt{b}(i) \in \mathcal{P} \quad \rightarrow \quad \mathcal{P} \setminus \{\mathtt{f}\} \cup \{\mathtt{b}(\mathtt{l})\}$

$\quad \mathtt{f}_1 \triangleq \forall i \in [\mathtt{l}, \mathtt{v}) : \mathtt{b}(i), \mathtt{f}_2 \triangleq \forall i \in [\mathtt{v}+\mathtt{1}, \mathtt{u}) : \mathtt{b}(i) \in \mathcal{P}$

$\qquad \rightarrow \quad \mathcal{P} \setminus \{\mathtt{f}_1, \mathtt{f}_2\} \cup \{\forall i \in [\mathtt{l}, \mathtt{u}) : \mathtt{b}(i)\}$

$\quad \mathtt{x}_1 - \mathtt{x}_2 \leq v_1, \mathtt{x}_2 - \mathtt{x}_3 \leq v_2, \mathtt{x}_1 - \mathtt{x}_3 \leq v_3 \in \mathcal{P}$

$\qquad \rightarrow \quad \textit{if } v_1 + v_2 \leq v_3 \textit{ then } \mathcal{P} \setminus \{\mathtt{x}_1 - \mathtt{x}_3 \leq v_3\} \textit{ else } \mathcal{P}$

Fig. 8. The simplification for the candidate preconditions. The function $s(\mathtt{b})$ returns a set whose constraints are the conjuncts in `b`.

can be eliminated from the set. If `false` appears as an atom, then there is no way to satisfy the precondition. `false` is the identity for disjunction, so it can be cancelled. In general the language short-cutting disjunction || is not commutative, but in our simplification procedure it can be moved to the front position (to enable the previous two rules). If an expression `b` is already in $\mathcal{P}$ with no antecedent, we can safely remove all the preconditions where `b` appears as a consequence. When `b` is implied by some condition and by its negation, then we can simply remove the conditions (we found this being a very common case in practice, when the precondition does not depend on the paths through a conditional statement). For remaining pairs with the same conclusion, we only retain the (disjunctive) precondition with fewer hypotheses. As for quantified facts, we remove those that boil down to a singleton and we merge together consecutive intervals. Finally, we remove from the difference constraints those that are redundant. The simplification should be iterated to a fixpoint: $\mathsf{Simplify}(\mathcal{P}_{\mathbb{A}}) \triangleq \mathsf{simpl}^*(\mathcal{P}_{\mathbb{A}})$.

7 Inter-Procedural Precondition Inference

The inter-procedural precondition inference algorithm is shown in Fig. 9. The input program `P` can either be a complete program or a library. We assume that each procedure `m` has an initial set of preconditions $\mathsf{PreconditionsOf}_0(\mathtt{m})$, which can be empty. The set `Next` contains the procedures to be analyzed (continuation set). It is initialized with all the procedures in the program `P`.

In the loop body, we first pick a procedure `m` from the continuation set. We leave the implementation of the policy PickOneProcedure as a parameter of the inference algorithm. For instance PickOneProcedure may be the bottom-up traversal of `P`'s call graph.

```
Next $\leftarrow$ ProceduresOf(P)
while Next $\neq \emptyset$ do
    m $\leftarrow$ PickOneProcedure(Next)
    $\mathbb{A} \leftarrow$ SpecificationFor(m, Clousot)
    $\mathcal{P}_{\mathbb{A}} \leftarrow$ InferPrecondition($\mathbb{A}$, Clousot)
    $\overline{\mathcal{P}}^{m}_{\mathbb{A}} \leftarrow$ Simplify(PreconditionsOf(m) $\cup\, \mathcal{P}_{\mathbb{A}}$)
    if $\overline{\mathcal{P}}^{m}_{\mathbb{A}} \neq$ PreconditionsOf(m) then
        PreconditionsOf(m) $\leftarrow \overline{\mathcal{P}}^{m}_{\mathbb{A}}$
        Next $\leftarrow$ (Next $\setminus$ {m}) $\cup$ callersOf(m)
    else
        Next $\leftarrow$ Next $\setminus$ {m}
    end if
end while
```

Fig. 9. The inter-method preconditions inference algorithm.

The function SpecificationFor returns the residual specification $\mathbb{A}$ of m by running Clousot to discharge as many proof obligations as possible. If $\mathbb{A} = \emptyset$, then we say that the preconditions PreconditionsOf(m) are *sufficient* to ensure that each proof obligation in m is either always correct or always wrong. The analyzer Clousot is left as a parameter.

We infer the set $\mathcal{P}_{\mathbb{A}}$ of necessary preconditions from the assertions in $\mathbb{A}$. The intra-method inference algorithm InferPrecondition is one of the previously described analyses such as APPA, CPPA, or QPA.

Next, the function Simplify removes redundant preconditions so as to obtain an approximate minimal set $\overline{\mathcal{P}}^{m}_{\mathbb{A}}$ of conditions as described in Sec. 6.

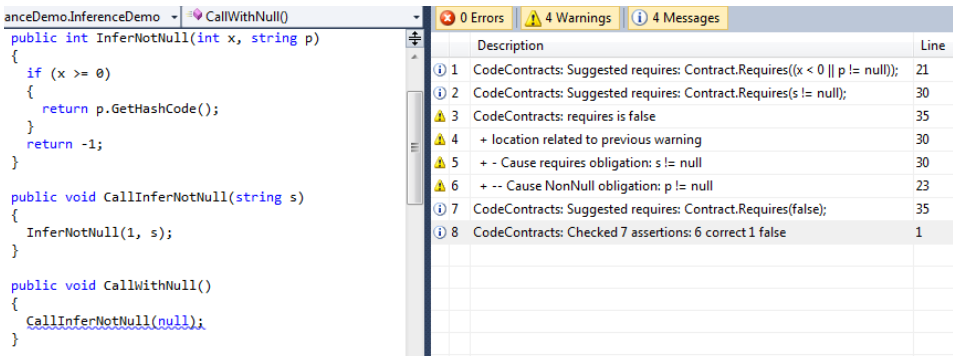
Finally, if we discovered new preconditions for m, we add them to the set of its preconditions, and we update the continuation set by adding all the callers of m, which we must re-analyze due to the stronger proof-obligations at these call-sites. In case we discovered no new precondition, we simply remove m from the continuation set.

The inter-method inference algorithm in Fig. 9 is a least fixpoint computation on the abstract domain $\langle \mathsf{ProceduresOf}(\mathtt{P}) \rightarrow \wp(\mathbb{B}), \dot{\Leftarrow} \rangle$, where $\mathbb{B}$ is the set of side-effect free Boolean expressions and $\dot{\Leftarrow}$ is a sound abstraction of the pointwise inverse logical implication. In the presence of (mutual) recursion the algorithm may not terminate: for instance it may infer the increasing chain of preconditions $\{0 < \mathtt{a}\} \dot{\Leftarrow} \{1 < \mathtt{a}\} \dot{\Leftarrow} \ldots$ To enforce convergence, a dual widening operator should be used: the simplification is incomplete so it does not solve the convergence problem in the abstract even in case of convergence in the concrete. Easy dual widening operators are either bounding the maximum number of times a method is analyzed or the maximum cardinality of PreconditionsOf(m) or both. Ignoring preconditions is safe: intuitively it means that fewer checks are pushed up in the call stack but warnings are still reported in the callee.

In practice, we can stop inferring new necessary preconditions at any point. The remaining methods in Next then simply need to be checked again (without inferring new preconditions) to obtain the final set of warnings.

```
void f(string x, string y) {          void g(string u, string v) {
  Assert(x != null);                    Assert(v != null);
  g(x, y);                              f(u, v);
}                                     }
```

Fig. 10. An example of inference of preconditions for mutually recursive methods requiring a fixpoint computation.



```
public int InferNotNull(int x, string p)
{
  if (x >= 0)
  {
    return p.GetHashCode();
  }
  return -1;
}

public void CallInferNotNull(string s)
{
  InferNotNull(1, s);
}

public void CallWithNull()
{
  CallInferNotNull(null);
}
```

	Description	Line
1	CodeContracts: Suggested requires: Contract.Requires((x < 0 || p != null));	21
2	CodeContracts: Suggested requires: Contract.Requires(s != null);	30
3	CodeContracts: requires is false	35
4	+ location related to previous warning	30
5	+ - Cause requires obligation: s != null	30
6	+ -- Cause NonNull obligation: p != null	23
7	CodeContracts: Suggested requires: Contract.Requires(false);	35
8	CodeContracts: Checked 7 assertions: 6 correct 1 false	1

Fig. 11. A screenshot of the error reporting with the precondition inference.

Example 8. Let us consider the two mutual procedures in Fig. 10. Ignore non-termination: we choose a minimalistic example to illustrate the inter-method fixpoint computation. Let us suppose `f` is the first method to be picked up. The intra-method precondition inference algorithm obtains $\mathsf{PreconditionsOf}(\mathtt{f}) = \{x! = \mathtt{null}\}$. The preconditions for `g` are then $\{u! = \mathtt{null}, v! = \mathtt{null}\}$. The procedure `f` is a caller of `g`, so it is added to the continuation set. The re-analysis is enough to reach the fixpoint: $\mathsf{PreconditionsOf}(\mathtt{f}) = \{x! = \mathtt{null}, y! = \mathtt{null}\}$. □

7.1 Provenance

Each precondition `p` in $\mathcal{P}_{\mathbb{A}}$ originates from at least one failing proof obligation $\langle \mathtt{c}, \mathtt{b} \rangle \in \mathbb{A}$. We can construct a provenance relation $\mathtt{p} \gg \mathtt{b}$, with the intuitive meaning that if `p` does not hold at the method entry, then `b` will fail later. We use the provenance chain $\mathtt{p}_{n-1} \gg \cdots \gg \mathtt{p}_0 \gg \mathtt{b}$ to report an inter-method error trace to the user. Furthermore, we can suppress the warning for `b` if we detect that `p` is also sufficient to prove `b` safe, *i.e.*, `p` holds at the entry point if and only if `b` holds at program point `c`. This is the case when at least *one* of these conditions holds: (i) the method `m` does not contain loops; (ii) `p` is inferred using APPA (Sec. 5.1); (iii) `p` is inferred using CPPA (Sec. 5.2) *and* no loop is encountered in the path between `entry` and `c`. Essentially, if we detect that the generated necessary precondition `p` is also sufficient to discharge `b`, we can push the *full* burden of the proof to the callers of `m`. Otherwise, we report the warning to the user and we propagate `p` to the callers, as failure to establish it will definitely cause an error in `b`: by propagating `p` we make explicit to the callers thay they *must* establish `p` before calling `m`.

8 Experience

We implemented the analyses described above in `cccheck`, an industrial-strength static analyzer and contract checker for .NET bytecode based on abstract interpretation. It uses abstract interpretation to infer facts about the program and to discharge the proof obligations. `cccheck` performs a modular assume/guarantee analysis: for each method it assumes the precondition and it asserts the postcondition. At method calls, it asserts the precondition, and it assumes the postcondition. It performs a simple postcondition inference to propagate whether a method returns a non-null value and the expression returned by getters. The necessary precondition inference is enabled by default in every run of the analyzer, and used by our customers since June 2011 on a daily basis. The kind of intra-method precondition inference algorithm can be selected by a command line switch. The all-path precondition analysis (APPA) is the default. We implemented the analyses faithfully respecting the formalization in the previous sections. The only differences are in the quantified precondition analysis: (i) we restrict it to arrays of objects (instead of collections of generic types); (ii) the only assertions we check for are not-null; and (iii) the quantified preconditions are suggested to the user but not propagated to the callers (yet).

The main motivation for this work was to help our users getting started with `cccheck`, by suggesting preconditions, so that users can simply add them to their code (or automatically with support from the IDE).

In an early stage of this work, we had a simple analysis to infer *sufficient* preconditions (Sect. 3). Essentially, if we could prove that the value of an unproven proof obligation was unchanged from the entry point, then we suggested the corresponding expression as a precondition. This is similar to the ESC/Java `/suggest` switch [18]. The problem with this naïve approach was that it did not take into account tests and different execution paths and so can also eliminate good runs. The result was confusing for our customers, who filed several bug reports about "wrong" suggestions (essentially preconditions that were too strong according to the user). For instance, in the code of Fig. 11 our old analysis would have produced the too strong precondition `p! =null` for the method `InferNotNull`.

Necessary precondition inference reduces the warning noise of the analyzer and raises the automation bar by providing inter-method propagation of preconditions. If a necessary precondition can be inferred from a proof obligation and `cccheck` determines that it is also sufficient then the warning can be omitted, i.e., the full burden of establishing it is pushed to the callers. The check for sufficiency can be: (i) by construction, *e.g.*, when APPA is used; (ii) inferred from the analysis, *e.g.*, by detecting that the dual widening in CPPA, if any, has introduced no loss of precision; (iii) by re-analysis, the generated precondition is injected in the method which is then reanalyzed. In `cccheck` we use (i) and (ii). We instrumented `cccheck` to perform (iii) to collect the data for Sec. 8.3.

Example 9. Let us consider the screenshot in Fig. 11 showing the result of a run of `cccheck` inside Visual Studio. On the right is the list of suggestions and warnings produced by cccheck for the code on the left. The analyzer performs a bottom-up analysis, based on a precomputed approximation of the call graph. First it analyzes InferNotNull. It determines that when x is non-negative and p is null then a runtime error may occur. It infers a necessary precondition (message #1) which is also sufficient—no approximation comes from loops. Therefore the generated precondition fully captures the safety requirement and no warning is issued to the user for p's dereference since the proof burden is pushed to the callers—in our example CallInferNotNull. This method does not fully establish precondition #1, and our inference computes another necessary precondition #2. Please note that this precondition is simpler than #1 since our simplification procedure removed the trivial disjunct $1 < 0$. Precondition #2 happens to also be sufficient in this case as there are no loops. cccheck detects this fact, and no warning is issued in method CallInferNotNull. Instead, precondition #2 is propagated to the call-site within CallWithNull. There, cccheck determines that the call does not satisfy the inferred necessary precondition of CallInferNotNull. It reports the error message (#3) as well as the inter-method provenance trace (messages #4 . . . #6). The generated precondition #7 encodes the fact that all invocations of CallWithNull will definitely cause an error. □

8.1 Benchmarks

We report our experience on two different sets of benchmarks. The first one contains industrial libraries without existing contracts. We have chosen the latest versions of mscorlib.dll, System.dll and System.Core.dll as they are the main libraries of the .NET framework, and System.Data.dll because, in our experience, it causes trouble for static analyzers. The first three libraries contain the most common shared functionalities of the .NET framework (system type definitions, collections, reflection, cryptographic and communication primitives, etc.). The last one contains common code to access data from diverse sources. cccheck analyzes bytecode, so we can use it directly on the shipped binaries. The libraries are in every copy of Windows under the directory Windows/Microsoft.Net/Framework/. For our experiments we run cccheck with the default checks: non-null, array-out-of bounds obligations and contracts. Since the version of the libraries we analyzed contained no contracts, the only contracts are the inferred necessary preconditions propagated by cccheck itself.

The second benchmark is the open-source C# Facebook SDK which is available for download at facebooksdk.codeplex.com. It contains a set of libraries to help .NET programmers (Windows, Silverlight, Windows Phone 7, etc.) integrate their application with Facebook. We selected it because the code base is almost completely annotated with contracts. In our experiments, we opened the solution containing the SDK, built the source as-it-is and let cccheck run with the same settings as in the distribution: the collected proof obligations are non-null, array-out-of bounds, arithmetic errors, redundant assumptions detection, and the explicit contracts. We only added switches to force the analyzer to collect the data reported in the tables.

	All-paths APPA			Conditional-path CPPA			QPA
Library	$\mathcal{P}_{\mathbb{A}}$	$\overline{\mathcal{P}}^m_{\mathbb{A}}$	% rem.	$\mathcal{P}_{\mathbb{A}}$	$\overline{\mathcal{P}}^m_{\mathbb{A}}$	% rem.	$\forall$
`mscorlib`	5133	3437	33.04%	8756	6564	25.03%	36
`System`	4409	3446	21.84%	6709	5533	17.53%	9
`System.Core`	3202	2197	31.39%	4723	3744	20.73%	32
`System.Data`	5899	3563	39.60%	8435	5642	33.11%	11
Total	18643	12643	32.18%	28623	21483	24.94%	88
`Facebook`	146	119	18.49%	171	145	15.20%	1
`Facebook.Web`	53	53	0.00%	86	86	0.00%	0
`Facebook.Web.Mvc`	49	31	36.73%	25	10	60.00%	0
Total	248	203	18.15%	282	241	14.54%	1

Fig. 12. The number of inferred preconditions, for APPA, CPPA, and QPA, and the percentage of redundant preconditions removed by the simplification routine Simplify.

8.2 Inferred Necessary Preconditions

The table in Fig. 12 reports the number of necessary preconditions inferred for the benchmarks for all three analyses (APPA, CPPA, and QPA), when the fixpoint of the inter-method inference algorithm has been reached.

For the system libraries, the all-paths analysis (APPA) infers $18,643$ necessary preconditions. The simplification step removes more than 32% of the candidates, resulting in $12,643$ necessary preconditions that are suggested and propagated. For the Facebook SDK, APPA infers 248 candidates, filtering only 45.

The conditional-path analysis (CPPA) infers roughly 69% additional necessary preconditions than APPA for the system libraries but only 18% more for the Facebook SDK. The price to pay for the more refined analysis is time: in our experience CPPA can be up to 4 times slower than APPA. However, at worst the total inference time (including simplification) is less than 4 minutes for very complex libraries (`System.Data`). Otherwise the overall running time is on the order of tenths of seconds.

We manually inspected the necessary preconditions inferred for the Facebook SDK to check whether the simplification algorithm left any redundancy. We found only one redundant precondition, inferred by CPPA for `Facebook.Web`.

As one may expect, we inferred fewer universally quantified necessary preconditions. We inspected the 36 quantified necessary preconditions inferred for `mscorlib.dll`. It turns out that the analysis inferred conditions that at first looked suspicious. After a deeper investigation, we found that the analysis was right. However, it is more difficult to judge whether the analysis missed necessary preconditions it should have inferred. So, we inspected the proof obligations for the Facebook SDK. In `Facebook` we found only two proof obligations requiring a quantified contract. However, the needed contracts are not preconditions but an object invariant and a postcondition. The other two libraries make little use of arrays, so there was nothing interesting to be inferred there. Overall, we found the quantified necessary precondition analysis precise and fast enough.

	# of methods						precision	# m.
		at least	inferred	inferred	%	%	improve-	with 0
Library	total	one warn.	nec. pre.	suff. pre.	inferred	suff.	ment	warns
`mscorlib`	21226	6663	2765	1519	41.50%	22.80%	7.15%	16082
`System`	14799	5574	2684	1378	48.15%	24.72%	9.31%	10603
`System.Core`	5947	2669	1625	765	60.88%	28.66%	12.8%	4043
`System.Data`	11492	4696	2388	1152	50.85%	24.53%	10.02%	7948
Total	53464	19602	9462	4814	48.27%	24.56%	9.00%	38676
`Facebook`	455	186	111	93	59.68%	50.00%	20.43%	362
`Facebook.Web`	194	57	30	18	52.63%	31.58%	9.27%	155
`Facebook.Web.Mvc`	92	40	29	26	72.50%	65.00%	28.26%	78
Total	741	283	170	137	60.07%	48.41%	18.48%	595

Fig. 13. The experimental results for the all-paths precondition analysis (APPA).

	# of methods						precision	# m.
		at least	inferred	inferred	%	%	improve-	with 0
Library	total	one warn.	nec. pre.	suff. pre.	inferred	suff.	ment	warns
`mscorlib`	21226	7107	4062	1811	57.15%	25.48%	8.53%	15930
`System`	14799	5759	3546	1576	61.57%	27.37%	10.64%	10616
`System.Core`	5947	2740	2104	810	76.79%	29.56%	13.62%	4017
`System.Data`	11492	4824	3280	1292	67.99%	26.78%	11.24%	7960
Total	53464	20430	12992	5489	63.59%	26.87%	10.26%	38523
`Facebook`	455	186	130	92	69.89%	49.46%	20.22%	361
`Facebook.Web`	194	110	80	61	72.73%	55.45%	31.44%	145
`Facebook.Web.Mvc`	92	23	8	5	34.78%	21.74%	5.43%	74
Total	741	319	218	158	68.34%	49.53%	21.32%	580

Fig. 14. The experimental results for the conditional path precondition analysis (CPPA).

8.3 Quality of the Inferred Preconditions

We are left with the problem of judging the *quality* of the inferred necessary preconditions. Counting the number of inferred preconditions is not a good measure of success. Measuring the number of warnings without inference and with inference is a better approach, but can also be misleading for the following reason: if a method contains a warning and is called by n other methods, then if that single warning can be turned into a necessary precondition, it potentially results in n warnings at all call-sites. We decided to measure how the inference of necessary preconditions reduces the number of methods for which we report warnings. If we reduce the number of methods with warnings, we say that we improved the precision of our analyzer `cccheck`.

The tables in Fig. 13 and Fig. 14 report the effects of the all-path (APPA) and the conditional-path (CPPA) analyses on the precision of `cccheck`. For each library we report (i) the total number of methods analyzed; (ii) the number of methods on which `cccheck` originally reports at least one warning (without any inference); (iii) the number of methods for which `cccheck` infers at least one necessary precondition; and (iv) the number of methods for which the necessary preconditions were also sufficient to reduce the warnings in that method to zero.

The next three columns indicate (i) the fraction of methods with inferred necessary preconditions; (ii) the fraction of these for which the inferred preconditions are also sufficient; and (iii) the precision improvement as an increase in the number of methods with zero warnings. The last column contains the final total number of methods with 0 warnings, *i.e.* the methods which either did not require the inference of any precondition, *plus* the methods for which the inferred necessary precondition is sufficient. To check whether the necessary preconditions are also sufficient, we check: (i) that we inferred some necessary precondition for the method; and (ii) that $\mathbb{A} = \emptyset$ after re-analysis.

The conditional-path precondition analysis (CPPA) infers far more preconditions than APPA, and in general those preconditions are also more complicated, because they can be disjunctive. As a consequence, it is not a surprise that the final number of methods with zero warnings is smaller in the case of CPPA: the additional warnings are generated by the propagated inferred preconditions that cannot be proven by `cccheck` at call-sites.

In the framework libraries we were able to infer a necessary precondition for 48% of methods with APPA and for 63% of methods with CPPA. Interestingly, the necessary preconditions turned out to be also sufficient in 25% of methods for either analysis. The precision is even better for the Facebook SDK where we inferred necessary preconditions for more than 60% of methods. Additionally, the necessary preconditions where sufficient to prove the method correct in almost 50% of cases! We manually inspected the methods with remaining warnings. These resulted from missing contracts, *e.g.* postconditions for interface calls and abstract methods and object invariants.

Overall, precondition inference improved the precision of `cccheck` by roughly 10% for the framework assemblies and roughly 20% for the Facebook SDK.

9 Related Work

Our objectives are hardly comparable with those of unsound tools like PREfix [4] or its lightweight version PREfast which filter out potential errors, bug-finding tools like SAGE [20] that extends systematic dynamic test generation with formal methods, property checking tools combining static analysis and testing like YOGI [30], or verification environments like VCC [14] because an unsound assertion or a series of failing tests cannot be used as formal specifications and automatic inference of program properties is much more difficult than static verification of given properties annotating programs. However, automatically inferred necessary preconditions would definitely be useful for all these tools.

The closest related work we are aware of is Success typings [26], where types of function arguments are used instead of first-order formulae to express preconditions. Similar to our work, success typings capture as types, the conditions under which a function will definitely cause a runtime type error. Their analysis is limited to runtime type errors as opposed to general assertions, and appears to be used only to detect definite bugs in an untyped language. In addition, our approach can be used to reduce the annotation burden in verification, and the expressiveness of our preconditions is more general.

Most of the work on precondition inference focuses on the inference of *sufficient* preconditions for the partial correctness of a module, *e.g.*, [2,5,9,29,31], even if it is never explicitly expressed in those terms. We have discussed in Sec. 3 the drawbacks of using sufficient precondition inference in an automatic analysis setting. We are not aware of papers on sufficient precondition inference experimentally validating the inferred preconditions: (i) at call-sites; and (ii) on as large a scale as we did.

The related problem of generating procedure summaries (to scale up whole program analyses) received much attentions. Summaries can either be obtained via a combination of heuristics and static analyses [17,23], or via firm semantic and logical grounds, *e.g.*, [6,9,22]. However, the practical effectiveness of the so-inferred preconditions is still unclear: *e.g.*, [6] reports the inference of thousands of contracts but their quality—for instance to prove a property of interest like memory safety—is unknown.

Some of these approaches are also less modular than ours. E.g., the Houdini approach [17] starts out by adding a set of candidate preconditions (generated from a template) to all methods and then uses repeated sound modular reasoning to reject candidates that lead to unproven assertions. This approach produces preconditions that are non-necessary (no assertion would be triggered if violated). Worse, the set of preconditions produced for a method depends on the contexts that call the method. The fewer such contexts exist, the stronger the inferred precondition (in the worst case *false*). Our approach is more modular. The necessary preconditions inferred for a method do not depend at all on how the method is called.

Some authors focused on inferring preconditions of a fixed template [21,34] or summaries on abstract domains of finite height [33,36]. Those approaches to precondition inference inherit the problems of the techniques they are based on. Templates require too much user attention (to provide the template), are brittle, and do not scale up. Finite height abstract domains are not powerful enough [8]. We do not make any of those hypotheses in this work.

SnuggleBug [7], the dual analysis of [32], as well as other authors [19,25,35] use under-approximating symbolic backwards or dynamic analyses to find bugs. Our work is different in that we start out with a program verification problem with a number of unproven assertions. We use necessary preconditions to reduce the annotation burden experienced by the programmer. Using our static analysis, we can tell when a method has no more errors, whereas bug finding cannot. At the same time, our approach can also expose definite bugs, as shown in the scenario of Fig. 11. A main difference of our work with all the above is the handling of loops via fixpoints rather than some finite, partial unrolling.

For instance, with a (minor) modification of the universally quantified forward analysis, the implementation infers the necessary precondition $\texttt{newCarsOnly} \Rightarrow \forall \texttt{x} \in \texttt{c} : \texttt{x.getYear}() = 2009$ for the running example of [7], requiring that all cars in a list are manufactured in 2009. Snugglebug unrolls loops once or twice and will find violations only if it can find a list where the first or second car has the wrong date, but not the third.

Necessary preconditions are needed to define the problem of extract method with contracts [13]. They can be extended for the inference of necessary object invariants [1] and to propose automatic and verified code repairs [27].

Finally, our inferred necessary preconditions are human readable and can be persisted into code as documentation and to help future analysis runs, whereas the intermediate state of bug finding tools is rarely in a form that would make it useful for human consumption.

10 Conclusions

We presented the design and implementation of a system for the inference of *necessary* preconditions. We illustrated the algorithm for the inter-procedural inference. The algorithm is parameterized by the static analyzer used to discharge the proof obligations, the intra-method inference analysis, and the candidate precondition simplification routine. We improved the intra-method analyses of [11] by refining them with the invariants inferred by the static analyzer. We have implemented the inference in `cccheck`, an industrial static contract checker, and evaluated the analysis on real industrial code. We showed that our simplification algorithm, even if not complete, performs very well in practice by removing up to 33% of redundant preconditions and finding only one case in which a redundant precondition was not removed. We were able to infer necessary preconditions for up to 60% of methods reporting some warning. We validated the quality of the inferred preconditions by checking whether they were also sufficient to remove *all* warnings in a method (not only the warning they originated from). This was the case for 25% of methods with warnings. Overall, the necessary precondition inference has a large positive impact on `cccheck`, by improving its precision (in terms of methods with no warnings) up to 21%.

Looking forward, we want to investigate necessary preconditions inference for program optimization (by factoring and pushing up inevitable checks) and extend the approach to the inference of necessary postconditions (by pushing the assertions back to method calls).

Acknowledgments Leonardo De Moura for the discussions on simplification. Work supported in part by the National Science Foundation under Grant No. 0926166.

References.

[1] M. Bouaziz, M. Fahndrich, and F. Logozzo. Inference of necessary field conditions with abstract interpretation. In *APLAS*, LNCS, 2012.

[2] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI*, pages 46–55, 1993.

[3] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[5] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, pages 402–418, 2007.

[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. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, pages 363–374, 2009.
[8] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, pages 269–295, 1992.
[9] P. Cousot and R. Cousot. Modular static program analysis. In *CC 2002*, pages 159–178, 2002.
[10] P. Cousot and R. Cousot. An abstract interpretation framework for termination. In *POPL*, pages 245–258, 2012.
[11] P. Cousot, R. Cousot, and F. Logozzo. Contract precondition inference from intermittent assertions on collections. In *VMCAI*, pages 150–168, 2011.
[12] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.
[13] P. Cousot, R. Cousot, F. Logozzo, and M. Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts. In *OOPSLA*. ACM, 2012.
[14] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*, pages 429–430, 2009.
[15] I. Dillig, T. Dillig, and A. Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *SAS*, pages 236–252, 2010.
[16] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, pages 10–30, 2010.
[17] C. Flanagan and K. R. L. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, pages 500–517, 2001.
[18] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, pages 234–245, 2002.
[19] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
[20] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.
[21] S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, pages 120–135, 2009.
[22] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, pages 253–267, 2007.
[23] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, pages 232–241, 2006.
[24] C. A. R. Hoare. Algorithm 63: Partition. *Communications of the ACM*, 4(7):321, 1961.
[25] J. Hoenicke, K. R. M. Leino, A. Podelski, M. Schäf, and T. Wies. It's doomed; we can prove it. In *FM*, pages 338–353, 2009.
[26] T. Lindahl and K. F/ Sagonas. Practical type inference based on success typings. In *PPDP*, pages 167–178. ACM, 2006.
[27] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *OOPSLA*, pages 133–146. ACM, 2012.
[28] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.
[29] Y. Moy. Sufficient preconditions for modular assertion checking. In *VMCAI*, pages 188–202, 2008.
[30] A. V. Nori, S. K. Rajamani, S. Tetali, and A. Thakur. The yogi project: Software property checking via static analysis and testing. In *TACAS*, pages 178–181, 2009.
[31] A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *CAV*, pages 314–327, 2008.
[32] C. Popeea and W.-N. Chin. Dual analysis for proving safety and finding bugs. In *SAC*, pages 2137–2143, 2010.
[33] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.
[34] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, pages 223–234, 2009.
[35] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *ICSE*, 2011.
[36] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL*, pages 221–234, 2008.