Summary-Based Inter-Procedural Analysis via Modular Trace Refinement

Franck Cassez¹, Christian Müller², and Karla Burnett¹

¹ NICTA and UNSW, Australia
² TU Munich, Germany

Abstract

We propose a generalisation of trace refinement for the verification of inter-procedural programs. Our method is a top-down modular, summary-based approach, and analyses inter-procedural programs by building function summaries on-demand and improving the summaries each time a function is analysed. Our method is sound, and complete relative to the existence of a modular Hoare proof for a non-recursive program. We have implemented a prototype analyser that demonstrates the main features of our approach and yields promising results.

1998 ACM Subject Classification D.2.4 [Software Engineering] Software/Program Verification, D.3.1 [Programming Languages] Formal Definitions and Theory, F.3.1 [Logics and Meanings of Programs] Specifying and Verifying and Reasoning about Program

Keywords and phrases Program verification, Hoare Logic, Refinement, Automata

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2014.545

1 Introduction

Automated software verification has made tremendous progress in the last decade. Static analysis tools are routinely used to analyse source code and have revealed many subtle bugs.

We address the problem of designing a context-sensitive, scalable inter-procedural analysis framework. Our method is fully modular, and analyses each function without in-lining function calls but rather by using an input/output summary for the functions. This provides scalability. Context-sensitivity provides accuracy and is achieved by building function summaries in a top-down manner and being able to refine these summaries (on-demand) during the analysis. The result of our algorithm (when it terminates) is either a proof that a program is error-free or an inter-procedural counter-example that witnesses the error.

Our method is a modular inter-procedural extension of refinement of trace abstraction [13] and inherits the main features of this approach: it is sound and complete w.r.t. the existence of a modular Hoare proof for the non-recursive program and strictly more powerful than predicate abstraction refinement. Due to space limitation, technical proofs that were part of the submitted version are now omitted.

2 Example

We consider inter-procedural programs like $P_1$ in Listing 1. The variables in each function (or procedure) are either input (read-only), output or local (read-write) integer variables. The variable $m$ is local to main and $n$ is an output variable. The variables $p, q$ are input variables of inc and $r$ is the output variable. The semantics of a function call like $n = \text{inc}(1, m)$ (line 3) is that the left-hand-side variables ($n$) are assigned the values of the corresponding output variables ($r$) at the end of the computation of the callee (inc).
Functions can be annotated with \texttt{assume} statements (e.g., \texttt{m >= 1} holds after line 2) and \texttt{assert} statements which assert predicates that should not be violated, otherwise the program contains an error. The objective is to check whether a program contains an execution that violates an \texttt{assert} statement. A function \( f \) is mapped to a function automaton, \( A_f \), which is an extended control flow graph (CFG) with a single entry (green) and a single exit (red) node. To track \texttt{assert} statement violations across function boundaries in a modular manner, we introduce an extra output variable, \( \texttt{error} \), together with additional edges in the CFG of a function: i) a call like \( \texttt{n = inc(1,m)} \) is virtually expanded in \( (\texttt{error}, \texttt{n}) = \texttt{inc(1,m)} \) i.e., each function returns its internal error status; ii) each statement \( \texttt{assert(\phi)} \) is viewed as an if statement, such that if \( \phi \) does not hold the variable \( \texttt{error} \) is set to 1 and the control jumps to the exit node of the function; this is exemplified by edges 8 to 0 and 0 to 13 in automaton \( A_{\text{inc}} \); iii) as errors may occur during function calls, the status of the caller's \( \texttt{error} \) variable is checked after each function call: if it is set the control goes to the exit (red) node otherwise to the next instruction; this is exemplified by node E and edge E to 5 in automaton \( A_{\text{main}} \).

Program \( P_1 \) has two possible causes of error: one is to call \texttt{inc} with a negative value for parameter \( p \) (violating the assertion at line 8), and the other is to violate the assertion on \( n \) at line 4 in \texttt{main}.

The (partial) correctness of \texttt{inc} can be expressed by the Hoare triple\(^2\) \( \{\neg \texttt{error}_2\} \texttt{inc} \{\neg \texttt{error}_2\} \). To check whether the Hoare triple \( \{\neg \texttt{error}_2\} \texttt{inc} \{\neg \texttt{error}_2\} \) holds we try to find a counter-example trace: if we succeed we disprove the Hoare triple, if we fail the triple is valid. To do so, we view \( A_{\text{inc}} \) as a language acceptor with initial state \texttt{entry} (green) and final state \texttt{exit} (red) and the existence of a counter-example amounts to finding a feasible trace in \( L(A_{\text{inc}}) \), the language accepted by \( A_{\text{inc}} \). Determining whether such a feasible trace exists can be achieved using an iterative trace refinement algorithm \([13, 15]\): 1) Pick a trace\(^3\) \( w \in L(A_{\text{inc}}) \); 2) Add the pre/post conditions \( \neg \texttt{error}_2 / \texttt{error}_2 \) to the trace \( w \) and check whether

\begin{verbatim}
1 proc main() returns (n) {
2   assume(m >= 1);
3   n = inc(1, m);
4   assert(n >= 0);
5 }
6
7 proc inc(p,q) returns (r) {
8   assert(p >= 0);
9   if (p >= 1)
10      r = q + 1;
11  else
12      r = q;
13  endif;
14 }
\end{verbatim}

\(1\) We introduce one indexed \texttt{error} variable per function to simplify the technical developments. We can equivalently use a global \texttt{error} variable to track the error status of a complete program.

\(2\) We use \texttt{error} (resp. \( \neg \texttt{error} \)) as a shorthand for “every valuation such that \texttt{error} is 1 (resp. 0)”.

\(3\) Traces of length \( n \) are written \( a_1 \cdot a_2 \cdots a_{n-1} \cdot a_n \).
the extended trace \( w' = \neg \text{error}_2 \cdot w \cdot \text{error}_2 \) is feasible; 3) If it is, we have found a counter-example and the triple is not valid. Otherwise, we look for a new trace in \( L(A_{\text{inc}}) \setminus \{w\} \). If \( L(A_{\text{inc}}) \setminus \{w\} \) is empty, the triple is valid; otherwise we start again at step 1 using a refined language \( L(A_{\text{inc}}) \setminus \{w\} \) which does not contain \( w \). Notice that this process may not terminate. A key result of [13] is that, for each infeasible trace \( w \), a rejecting automaton, \( A(w) \), can be computed that accepts traces that are infeasible for the same reasons as \( w \). Thus, in the refinement step 3), we can remove all the traces accepted by automaton \( A(w) \) and not only \( \{w\} \).

The outcome of the iterative trace refinement algorithm (when it terminates) is either a counter-example path or a confirmation that a triple holds. Our first result (Section 4) is that, when we establish that \( \{P\} \not\{Q\} \) holds, we get a better triple \( \{P'\} \not\{Q'\} \) with \( P \Rightarrow P' \) (weaker assumption on input) and \( Q' \Rightarrow Q \) (stronger constraint on input/output relation). Our main result (Section 5) is the extension of the trace refinement approach [13] to check whether triples hold for inter-procedural programs without in-lining function calls.

The main idea of our extension is illustrated next. A function call is viewed as a standard instruction: the call \( r = f(m) \) defines a relation between the input variables \( m \) and the output variables \( r \). The only difference to a standard instruction is that we do not exactly know this relation, which is the strongest postcondition operator for the function.

This can be remedied as follows: for each function call to \( f \), we use a summary which is an over-approximation of the strongest postcondition of the function \( f \). A summary for \( \text{inc} \) could be \( p \geq 1 \Rightarrow r \leq q + 1 \) or, if we do not know anything about \( \text{inc} \), \( \text{True} \Rightarrow \text{True} \), which means that the output variables can be assigned any value. To determine the status of triple \( (H) \{\neg \text{error}_1\} \text{main} \{\neg \text{error}_1\} \), we try to find a witness trace in \text{main} invalidating it, i.e., starting in \( \neg \text{error}_1 \) and ending in \( \text{error}_1 \):

1. let \( w_1 = m \gg 1 \cdot \text{error}_1, n = \text{inc}(1,m) \cdot \text{error}_1 \) be a trace in \( L(A_{\text{main}}) \).
2. Using the semantics of each statement, and the over-approximate summary semantics \((\text{True}, \text{True})\) for the function call, check whether \( w_1' = \neg \text{error}_1 \cdot w_1 \cdot \text{error}_1 \) is feasible. It is and we get a witness assignment for the values of the variables in \( w_1' \) that implies a pre/postcondition \( \pi_1 : p = 1 \land q = 1 \land \neg \text{error}_2/\text{error}_2 \) for \( \text{inc} \) to make \( w_1' \) feasible.
3. To determine whether \( \pi_1 \) can be satisfied, we can establish the status of the opposite triple \( \{p = 1 \land q = 1 \land \neg \text{error}_2\} \text{inc} \{\neg \text{error}_2\} \).
   This triple holds and thus the corresponding witness pre/postcondition \( \pi_1 \) in \text{main} is infeasible. While establishing this, we have computed a stronger valid triple, \( \{p \geq 0 \land \neg \text{error}_2\} \text{inc} \{\neg \text{error}_2\} \) that can, from now on, be used as a valid summary \( (G_1, S_1) = (p \geq 0 \land \neg \text{error}_2, \neg \text{error}_2) \) for \( \text{inc} \).
4. We again check the feasibility of \( w_1' \), this time with the new summary \( (G_1, S_1) \). Using \( (G_1, S_1) \), \( w_1' \) becomes infeasible and thus \( w_1 \) can be ruled out.
5. We pick a different trace \( w_2 = m \gg 1 \cdot \text{error}_1, n = \text{inc}(1,m) \cdot \neg \text{error}_1 \cdot !((n \gg 0) \cdot \text{error}_1 = 1 \land \neg \text{error}_1 \cdot w_2 \cdot \text{error}_1 \) is feasible.
   It is provided the call to \( \text{inc} \) can realise an input/output constraint given by a witness assignment for \( w_2' \). Function \( \text{inc} \) cannot realise this witness assignment and the result of this check is a new valid triple for \( \text{inc} \), \( (G_2, S_2) = (p \geq 1 \land q \geq 1, r \geq q + 1) \).
6. We again check the feasibility of \( w_2' \) with \( (G_1, S_1) \) and \( (G_2, S_2) \). \( w_2' \) is now declared infeasible with \( (G_1, S_1) \) and \( (G_2, S_2) \) and this enables us to rule out \( w_2 \) in \text{main}.
7. There is only one trace left to explore in \text{main} but it cannot set \( \text{error}_1 \). The final result is a triple \( \{\neg \text{error}_1\} \text{main} \{\neg \text{error}_1 \land n \geq m + 1\} \) that is stronger than the initial one.

---

4 Verifying C-like programs is undecidable.
5 \( m \) and \( r \) are vectors of variables, however for clarity we omit vector notation.
3 Preliminaries

Programs are written in a simple inter-procedural programming language as commonly assumed [2, 18]. There are no pointers, no global variables, and we restrict to integer variables. This last restriction is not important as integer variables are expressive enough to encode a very large class of errors in imperative programs e.g., array-out-of-bounds, NULL pointer dereferences, etc. We assume a set of predicates over a set of variables e.g., in Quantifier-Free Linear Integer Arithmetic. Given a predicate $\varphi$, $\text{VAR}(\varphi)$ is the set of variables appearing in $\varphi$. We freely use the logical notation or set notation depending on which is best suited, e.g., given two predicates $P$ and $Q$, we use $P \land Q$ (logical and) or $P \cap Q$ (set intersection). $\text{False}$ corresponds to the empty set and $\text{True}$ to the set of all possible values.

The set of program statements $\Sigma$ is comprised of: (i) simple assignments e.g., $y = t$ where $y$ is a variable and $t$ a linear term on variables, (ii) assume statements which are predicates over the variables and (iii) function calls of the form $r_1, \ldots, rk = f(d_1, \ldots, d_n)$ where $f$ is a function and $r_1, \ldots, rk$ and $d_1, \ldots, d_n$ are the input and output variables.

Given a simple assignment $st$ and a predicate $\varphi$, $\text{post}(st, \varphi)$ is the strongest condition that holds after executing $st$ from $\varphi$. For an assume statement $st$, the semantics are $\text{post}(st, \varphi) = \varphi \land st$. The semantics of each function call are given by the strongest postcondition operator $\text{post}$ for the function (although we may not explicitly have it). The post operator extends straightforwardly to traces in $\Sigma^*$.

A trace $t$ satisfies a pre/post condition $(P, Q)$ if $\text{post}(t, P) \subseteq Q$. A trace $t$ is $(P, Q)$-feasible if $\text{post}(t, P) \cap Q \not\subseteq \text{False}$, otherwise it is infeasible. We let $\text{Infeas}(P, Q)$ be the set of traces over $\Sigma^*$ that are $(P, Q)$-infeasible. A trace $t$ is infeasible if it is $(\text{True}, \text{True})$-infeasible (or if it satisfies $(\text{True}, \text{False})$, otherwise it is feasible); $\text{Infeas}$ is the set of infeasible traces.

A trace automaton [13, 15] is a tuple $A = (L, \delta, L_{\text{init}}, L_{\text{exit}})$ where $L$ is a finite set of locations, $\delta \subseteq L \times \Sigma \times L$ is the transition relation, and $L_{\text{init}}, L_{\text{exit}} \subseteq L$ are the initial and final (accepting) locations. The language accepted by $A$ is $L(A)$.

A function $f$ is represented by a function automaton which is a trace automaton $A_f = (L_f, \delta_f, \{\text{init}_f\}, \{\text{exit}_f\})$. It is obtained from the CFG of $f$ by adding the edges setting the error variable to encode assert statement violations (see $A_{\text{main}}$ and $A_{\text{inc}}$ in Figure 1).

4 Checking Intra-Procedural Partial Correctness

We assume in this section that functions do not contain function calls. We show how to construct automata that accept traces that satisfy Hoare triples. This extends the results of [13]. A similar development is accounted for in [15] but we establish here a new useful result in Theorem 5. Given a trace automaton $A$ and two predicates $P, Q$ (over the variables of $A$), the Hoare triple $\{ P \} A \{ Q \}$ is valid iff $\forall t \in L(A), \text{post}(t, P) \subseteq Q$. Program (or function) correctness [13] is defined by: $\{ P \} f \{ Q \}$ is valid iff $\{ P \} A_f \{ Q \}$ is valid. The validity of a Hoare triple $\{ P \} A_f \{ Q \}$ can be expressed in terms of language inclusion:

$\triangleright$ Theorem 1. $\{ P \} A_f \{ Q \}$ is valid iff $L(A_f) \subseteq \text{Infeas}(P, \neg Q)$.

We also extend the notion of inductive interpolants [13] for infeasible traces to $(P, Q)$-interpolants for $(P, Q)$-infeasible traces. Let $t = st_1 \cdots st_k$ be a $(P, Q)$-infeasible trace. A sequence of predicates $I_0, I_1, \ldots, I_k$ is a $(P, Q)$-interpolant for $t$ if: 1) $P \implies I_0$, 2) $\forall 1 \leq i \leq k, \text{post}(st_i, I_{i-1}) \subseteq I_i$ and 3) $I_k \land Q = \text{False}$. For $t \in \text{Infeas}(P, Q)$, we let $\text{itp}_{P,Q}(t)$ be the set of $(P, Q)$-interpolants for $t$. For $t \in \text{Infeas}$, we let $\text{itp}(t)$ be the set of interpolants for $t$. By Craig’s interpolation theorem [10], we know that $\text{itp}(t) \neq \emptyset$. It follows that:
Lemma 2. If \( t \in \text{Infeas}(P,Q) \) then \( \text{itp}_{P,Q}(t) \neq \emptyset \).

Notice that the ability to compute actual interpolants (e.g., using SMT-solvers) is inessential as inductive interpolant can always be obtained using weakest preconditions for \( t \).

Let \( t = st_1 \cdots st_k \) be an infeasible trace and \( I = I_0, I_1, \ldots, I_k \) be an interpolant for \( t \).

The canonical interpolant automaton \([13]\) for \((t,I)\) is a trace automaton \( A_t^I = (L_T, \delta_T, \{\text{init}_T\}, \{\text{exit}_T\}) \). An important property of canonical interpolant automata is that they accept sets of infeasible traces:

Theorem 3 ([13]). If \( t \in \text{Infeas} \) and \( I \in \text{itp}(t) \) then \( L(A_t^I) \subseteq \text{Infeas} \).

We extend the definition of canonical interpolant automata to \((P,Q)\)-interpolant automata. Let \( t = st_1 \cdot st_2 \cdots st_k \in \text{Infeas}(P,Q) \) and \( I = I_0, I_1, \ldots, I_k \in \text{itp}_{P,Q}(t) \). Then \( t \) is also in \( \text{Infeas}(I_0,Q) \). Let \( t' = \text{assume}(I_0) \cdot t \cdot \text{assume}(Q) \) (in the sequel we write such a trace \( I_0 \cdot t \cdot Q \) omitting the \text{assume} statements). As \( t \in \text{Infeas}(I_0,Q) \), we have \( t' \in \text{Infeas} \) and moreover \( t' = \text{True}, I_0, \ldots, I_k, \text{False} \) is an interpolant for \( t' \). We can then build the canonical interpolant automaton \( A_{t'}^I = (L_{T'}, \delta_{T'}, \{\text{init}_{T'}\}, \{\text{exit}_{T'}\}) \) for \((t',I')\). We define the corresponding \((P,Q)\)-interpolant automaton for \((t,I)\) as the tuple \((A(P,Q)^I_t, L_{T'}, \delta_{T'}, \{\text{init}_{T'}\}, \{\text{exit}_{T'}\})\) where: 1) \( L_{T'} = L_T \setminus \{\text{init}_T, \text{exit}_T\} \), 2) \( L_{T'}^\text{init} = \{ \ell \in L_T \mid (\text{init}_T, P, \ell) \in \delta_T \} \) and \( L_{T'}^\text{exit} = \{ \ell \in L_T \mid (\ell, Q, \text{exit}_T) \in \delta_T \} \) and 3) \( \delta_{T'} = \delta_T \cup (L_T \times \Sigma \times L_T) \). \((P,Q)\)-interpolant automata accept sets of \((P,Q)\)-infeasible traces:

Theorem 4. If \( t \in \text{Infeas}(P,Q) \) and \( I \in \text{itp}_{P,Q}(t) \) then \( L(A(P,Q)^I_t) \subseteq \text{Infeas}(P,Q) \).

We can now introduce Algorithm 1, \text{Hoare}_1, that can establish the status of a Hoare triple. If the triple is valid, it returns a new stronger triple and otherwise a witness counter-example.

Algorithm 1 works as follows: the family of automata \( A_i, i \geq 1 \) accept only infeasible traces. If the condition of the while loop (line 1) is True, there is a trace \( t \) in the CFG of \( A_t \) that has not been declared \((P,\neg Q)\)-infeasible yet. This \((P,\neg Q)\)-infeasibility of trace \( t \) is investigated: if it is feasible, the triple \( \{P\} A_t \{Q\} \) does not hold and \( t \) is a witness (line 4). If it is infeasible, an interpolant automaton \( A_n \) is built from \( t \) (lines 6 to 9) and added to the family \( A_i, i \geq 1 \). If the condition of the while loop (line 1) is False, all the traces of the CFG of \( A_{\infty} \) have been declared \((P,\neg Q)\)-infeasible and hence \( \{P\} A_t \{Q\} \) holds. Moreover, as stated by Theorem 5 the interpolants collected at each step during the refinement can be used to build a stronger triple (line 10).

A triple \( \{P'\} \not\subseteq \{Q\} \) is stronger than \( \{P\} \not\subseteq \{Q\} \) if \( P \implies P' \) and \( Q' \implies Q \).

Line 9 of Algorithm 1 stores the interpolants each time a trace \( t \) is declared \((P,\neg Q)\)-infeasible. It collects the interpolants \( I_0 \) and \( I_k \) and stores them in the arrays \( P_n \) and \( Q_n \) (each interpolant automaton \( A_n \) is also stored). If the triple is valid, the interpolant automata \( A_t \) cover the set of traces of \( A_t \), and \( A_n \) satisfies the triple \( \cap_{i=1}^n P_i \cup \cap_{i=1}^n Q_i \).

Theorem 5. If Algorithm 1 terminates and returns \( \text{itp}(P',Q') \) then \( \{P'\} A_t \{Q'\} \) is valid and stronger than \( \{P\} A_t \{Q\} \).

If Algorithm 1 terminates it either returns: i) \( \text{path}(t) \), and then \( \{P\} \not\subseteq \{Q\} \) does not hold and \( t \) is such that \( \text{post}(t,P) \cap \neg Q \not\subseteq \text{False} \), or ii) \( \text{itp}(P',Q') \) then \( \{P'\} \not\subseteq \{Q'\} \) holds and \( P \implies P' \) and \( Q' \implies Q \). Claim i) holds because there are no function calls and the trace \( t \) selected at line 2 is such that \( \text{post}(t,P) \cap \neg Q \not\subseteq \text{False} \). As \text{post} is exact for statements which are not function calls, \( t \) is a counter-example. For ii), if \text{Hoare}_1 returns \( \text{itp}(P',Q') \), Theorem 5 holds proving correctness in this case. Termination of Algorithm 1 is of course not guaranteed as the verification problem for C-like programs is undecidable.
Algorithm 1: \( Hoare_1(A, P, Q) \)

**Input:** A function automaton \( A \), two predicates \( P \) and \( Q \).

**Result:** \( \text{itp}(P', Q') \) with \( P \implies P' \) and \( Q' \implies Q \) if \( \{P\} A \{Q\} \) holds,
path(\( t \)) with \( \text{post}(P, t) \) \( \cap \) \( \neg Q \subseteq \text{False} \) otherwise.

**Var:** \( \overline{A} \): arrays of interpolant automata, initially empty

\( P, Q \): arrays of predicates, initially empty

\( n \): integer, initially 0

1 while \( \mathcal{L}(A) \not\subseteq \bigcup_{i=1}^{n} \mathcal{L}(A_i) \) do

   /* There is a trace \( t \) from entry to exit in \( \mathcal{L}(A) \setminus \bigcup_{i=1}^{n} \mathcal{L}(A_i) \) */

2 Let \( t = s_1 \cdots s_k \in \mathcal{L}(A) \setminus \bigcup_{i=1}^{n} \mathcal{L}(A_i) \);

3 if \( \text{post}(t, P) \cap \neg Q \not\subseteq \text{False} \) then

   /* \( \text{post}(t, P) \subseteq Q \) does not hold, \( t \) is a counter-example */

4 return path(\( t \));

else /* \( t \) is \( (P, \neg Q) \)-infeasible, we refine and iterate */

6 Let \( I = I_0, \cdots, I_k \in \text{itp}_{P, \neg Q}(t) \);

7 Let \( n := n + 1; \)

8 Let \( A_n = A(P, \neg Q) I_k; \)

9 let \( P_n \) := \( I_0 \) and \( Q_n \) := \( I_k \);

10 return \( \text{itp}(\bigcap_{i=1}^{n} P_i, \bigcup_{i=1}^{n} Q_i) \);

However, similar to the trace refinement algorithm of [13], we can ensure incrementality and progress. Incrementality is relative to the Hoare triple we are checking, and means that once a \( (P, \neg Q) \)-interpolant automaton has been computed it holds during the call to \( Hoare_1(f, P, Q) \) and we never have to withdraw it. Progress is ensured because if we discover a \( (P, \neg Q) \)-infeasible trace \( t \) at the \( n \)-th iteration of the while loop, it is accepted by the corresponding automaton \( A_n \) and thus cannot be found in subsequent rounds. As pointed out in [13], soundness of the algorithm i.e., if it terminates and declares a program error-free, the program is actually error-free, is implied by Theorem 5. Completeness i.e., if a program is error-free, \( Hoare_1 \) can declare it error-free, holds (as usual) for a trivial reason [13] and we do not detail this.

5 Inter-Procedural Trace Refinement

Let \( f \) be a function with (formal) input parameters \( \overline{x} = x_1, \ldots, x_k \) and output parameters \( \overline{y} = y_1, \ldots, y_n \). We assume that input variables are not modified by a function and there are no global variables. A summary \( S(\overline{x}, \overline{y}) \) for \( f \) is a set of pairs of predicates \( (P_i(\overline{x}), Q_i(\overline{x}, \overline{y})))_{1 \leq i \leq n} \) where \( \varphi(\overline{x}) \) indicates that \( \text{VAR}(\varphi) = \{z_1, \cdots, z_n\} \). A summary can equivalently be viewed as a predicate \( \wedge_{i=1}^{n} P_i(\overline{x}) \implies Q_i(\overline{x}, \overline{y}) \).

The exact post operator is not explicitly available for functions. We want to over-approximate it while retaining enough information to prove a property of interest. We approximate the post operator for functions using summaries. A context \( \mathcal{C} \) is a mapping from functions to summaries such that for each function \( f \) in the program, \( \mathcal{C}(f) \) is a summary for

---

6 These limitations are not compulsory but are commonly admitted [2, 18].
The main difference between Algorithm 1 and Algorithm 2 is in how feasibility of a trace
we introduced so far are also valid for the
Algorithm 3. Algorithm 2 determines the status of a candidate triple
is checked using the summaries (call to
▶ Proposition 6. Let \((P, Q)\) be two predicates on the input and input/output variables of \(f\)
such that \(\{P\} A_f \{Q\}\) holds. Then \(\text{post}(r = f(m), \varphi) \subseteq \text{post}(C : f \mapsto \{(P, Q)\}, r = f(m), \varphi)\).

Proposition 6 generalises to summaries that are sets of pairs of predicates:
▶ Theorem 7. Let \(C\) be an over-approximation, \(f\) a function and \((P, Q)\) be two predicates
on the input and input/output variables of \(f\) such that \(\{P\} A_f \{Q\}\) holds. The context \(C'\)
defined by \(C'(h) = C(h)\) for \(h \neq f\) and \(C'(f) = C(f) \cup \{(P, Q)\}\) is an over-approximation.

We can now propose our new modular trace refinement algorithm (Algorithms 2 and 3).
The main difference between Algorithm 1 and Algorithm 2 is in how feasibility of a trace
is checked using the summaries (call to \(\text{Status}\) at line 3). This step is more involved
and is detailed in Algorithm 3. Algorithm 2 determines the status of a candidate triple
\(\{P\} A_f \{Q\}\) and either returns an inter-procedural (counter-example) path or a stronger
triple \(\{P'\} A_f \{Q'\}\). An inter-procedural path for a trace \(t\) of function \(g\) is inductively defined
by a mapping \(\text{path}\) such that:
- for statements \(st\) in \(t\) that are not function calls, \(\text{path}(st) = st\),
- for \(st\) a function call \(r = f(m)\), \(\text{path}(st)\) is an inter-procedural path for \(f\).

In Algorithm 2, we assume that the context variable \(C\) is a global variable and is initialised
with default summaries e.g., \((\text{True}, \text{True})\) for each function. In Algorithm 2, line 3, the call
to \(\text{Status}(t, P, \neg Q)\) (\(\text{Status}\) is defined in Algorithm 3) returns the status of \(t\): it is either
\((P, \neg Q)\)-feasible and \((\text{True}, \text{path}(t))\) is returned or \(C-(P, \neg Q)\)-infeasible and \((\text{False}, \perp)\) is
returned (\(\perp\) stands for the void path). \(\text{Hoare_2}\) is very similar to Algorithm 1 once the status
of a trace \(t\) is determined:
- if \(t\) is \(C-(P, \neg Q)\)-infeasible i.e., \(\text{Status}(t, P, \neg Q) = (\text{False}, \perp)\), then it is \((P, \neg Q)\)-infeasible
\((C\) is an over-approximation) and we can compute an interpolant automaton to reject
similar traces (lines 7 to 10 in Algorithm 2). Note also that the summary for the currently
analysed function is updated (line 11).
- otherwise \(t\) is \((P, \neg Q)\)-feasible i.e., \(\text{Status}(t, P, \neg Q) = (\text{True}, \text{path}(t))\) and \(t\) is an inter-
procedural counter-example.
Algorithm 2: $Hoare_2(A_t, P, Q)$

Global: $C$: context with summaries for each function, initially $(True, True)$

Input: A function automaton $A_t$, two predicates $P$ and $Q$.

Result: If $\{P\} A_t \{Q\}$ holds, $itp(P', Q')$ with $P \models P'$ and $Q' \models Q$.
Otherwise an inter-procedural path $\text{path}(t)$ with $\text{post}(P, t) \cap \neg Q \not\subseteq False$.

Local: $\mathcal{A}$: arrays of interpolant automata, initially empty
$P, Q$: arrays of predicates, initially empty
$n$: integer, initially 0

/* Main refinement loop */
1 while $\mathcal{L}(A_t) \not\subseteq \bigcup^n_{i=1} \mathcal{L}(A_i)$ do
   /* There is a trace from entry to exit in $\mathcal{L}(A_t) \setminus \bigcup^n_{i=1} \mathcal{L}(A_i)$ */
   Let $t = s_1 \cdots s_k \in \mathcal{L}(A_t) \setminus \bigcup^n_{i=1} \mathcal{L}(A_i)$;
   /* Determine the status of $t$ (and update summaries $C$) */
   $R, \text{path}(t) := \text{Status}(t, P, \neg Q)$;
   /* Status of $t$ is settled */
   if $R$ then
      /* $\{P\} A_t \{Q\}$ is not valid and $\text{path}(t)$ a counter-example */
      return $\text{path}(t)$;
   else
      /* $t$ is $(P, \neg Q)$-infeasible, we refine and iterate */
      $\mathcal{L} := I_0, \cdots, I_k \in itp_{P, \neg Q}(t)$;
      $n := n + 1$;
      $A_n := A(P, \neg Q)^{itp}_{L_{s_1}}$;
      $P_n := I_0$ and $Q_n := I_k$;
      /* $\{P\} A_t \{Q\}$ is valid. Add to $C$ and returns a stronger summary */
      $C := C(t) \cup \{(\bigcap^n_{i=1} P_i, \bigcup^n_{i=1} Q_i)\}$;
      return $itp(\bigcap^n_{i=1} P_i, \bigcup^n_{i=1} Q_i)$;

$\text{Status}(t, P, \neg Q)$ is defined in Algorithm 3 and determines the $(P, \neg Q)$-feasibility status of a trace $t$, and in doing so may recursively call Algorithm 2 (line 8). Algorithm 3 determines the status of a trace $t = s_1 \cdots s_k$ as follows:

- Function call statements are collected and stored into $\text{FCall}$ (line 2). Then $\text{path}$ is initialised with the default values for the statements that are not function calls (line 3).
- The $(P, \neg Q)$-feasibility status of $t$ is determined in an iterative manner:
  - if $t$ is $C(P, \neg Q)$-infeasible, the condition of line 5 is false and the else statement on line 16 is executed. This implies $t$ is $(P, \neg Q)$-infeasible as $C$ is an over-approximation, and we can return $(False, \bot)$.
  - if $t$ is $C(P, \neg Q)$-feasible, we obtain some before/after witness values for the variables for each function call and store them in pairs $(\nu_i, \mu_i)$, $i \in \text{FCall}$. The for-loop at line 8 checks each function call w.r.t. to the feasibility of its before/after witness values. This is done by recursively calling $Hoare_2$ (Algorithm 2) on the callees by claiming that the witness assignment is not realisable by the function. The result of these recursive calls to $Hoare_2$ are either a witness trace $\text{path}(u)$ or a pair of predicates $itp(P', Q')$. If we get a witness trace we store it in $\text{path}(st_i)$ (line 12), otherwise we do nothing (but the context $C$ has been updated by the call to $Hoare_2$).

$\triangleright$ Remark. One important feature of the algorithm to build the canonical interpolant automata [13, 15] is the ability to add back edges (thus defining loops) to the initial automaton that encodes the infeasible trace. An (back) edge labelled $st$ can be added from
Algorithm 3: Status(t, P, ¬Q)

Global: C: context with summaries for each function
Input: (False, ⊥) if t is C-(P, ¬Q)-infeasible and thus (P, ¬Q)-infeasible
       (True, path(t)) with path(t) a (P, ¬Q)-feasible full inter-procedural path.

1. Let t = s(t1s(t2 · · · stk);
2. Let FCall = {1 ≤ i ≤ k | sti is a function call};
3. foreach i ∈ {1, · · · , k} \ FCall do path(sti) := sti;
4. while True do
5.     if post(C, t, P) \ ¬Q ⊆ False then
6.         /* t is (P, ¬Q)-feasible under C */
7.         Let {(νi, µi), i ∈ FCall} be the set of witness before/after values;
8.         /* Check whether each function call step is feasible */
9.         foreach i ∈ FCall do path(sti) = ⊥;
10.    foreach i ∈ FCall(t) do
11.        Let sti be a call to f with f defined by f(ν); switch Hoare2(ν, A, ¬(µ = µ)) do
12.           case path(u)
13.               /* u s.t. post(u, ν) = = µ */
14.               path(sti) := path(u);
15.           case itp(P′, Q′)
16.               /* f satisfies {ν} Aν {¬(µ = µ)} */
17.               /* (P′, Q′) has been added to summary of f */
18.               if  \ not add ∪ {P′, Q′} \ has been added to summary of f then
19.                   /* t is (P, ¬Q)-infeasible under C and thus (P, ¬Q)-infeasible */
20.               return (False, ⊥);
The assumption that the post operator is exact for simple statements can be lifted while still preserving soundness. An over-approximation for post (e.g., for programs with non-linear assignments, we can compute a linear over-approximation) ensures soundness. However, we may return a witness counter-example which is infeasible, and get some false positives.

Finally, as trace refinement is strictly more powerful [13] refinement-wise than predicate abstraction refinement, we obtain a modular inter-procedural analysis technique that is strictly more powerful than predicate abstraction refinement based modular inter-procedural analysis [2, 18].

6 Implementation and Experiments

We have implemented Algorithms 2 and 3 in a prototype iProc (written in Scala). The input language of iProc is a simple inter-procedural language and we use SMT-INTERPOL [9] as the back-end solver to check feasibility and generate inductive interpolants when needed. We have implemented our own algorithm to build interpolant automata. An initial Hoare triple \{P\} main {Q} is specified using assume and assert (e.g., Program \(P_1\), Listing 1).

We have experimented with small test cases inspired from industrial case studies submitted by Goanna [8] customers and users. We focussed on array-out-of-bounds and NULL pointer dereferences detection as this can be encoded in our programming language with integers. We specifically analysed inter-procedural programs that were generating false positives with the tool Goanna. The results show that we correctly analyse all the programs removing all the false positives generated by the latest release of Goanna. This clearly demonstrates an improvement with regard to accuracy.

We are now building a more versatile version of our prototype to be able to parse and analyse C programs and properly demonstrate scalability compared to other tools. Extensions to support arrays and pointer aliasing [7] are currently being investigated.

7 Related Work

Algorithms for inter-procedural (data flow) analysis of imperative programs can be traced back to 1978 with the seminal work of Sharir and Pnueli [19], and later by Reps et al. [17]. However, practical techniques and tools have only been discovered in the last decade. SLAM [4] is certainly the best known tool and has been successfully applied to large case studies (e.g., checking violations of API rules in device drivers.) It relies on powerful automated predicate abstraction refinement techniques. BLAST [5] and CPAchecker [6] have been successfully applied to medium size projects. The previous tools perform predicate abstraction (and refinement) rather than trace abstraction (and refinement), and to the best of our knowledge they do not fully support modular inter-procedural computation of increasingly precise summaries, but rather perform (some form of) function call in-lining.

The use of interpolants to extract summaries has been the subject of some recent papers. Two approaches are close in spirit to our work: WHALE [2] and FUNFROG [18]. There are fundamental differences between the algorithm in WHALE and FUNFROG and ours. WHALE builds under-approximations of functions which has a major drawback: an already computed summary is valid provided the summaries of other functions are valid; if it turns out that an existing summary is invalidated (which can happen as only an under-approximation of a function has been explored), all the dependent computed summaries are invalidated as well. FUNFROG [18] is based on bounded model-checking and thus proves properties of functions upto a bounded unrolling for loops and recursive calls. The summaries computed
by FunFrog are thus valid only for the bounded unrolling of the function (e.g., this might prevent this approach from discovering loop invariants). Moreover, the computation of the summaries themselves is not modular: if a trace is feasible using the currently available summaries, and must be further investigated, this is done by in-lining suspicious calls to check whether they are actually feasible. SMASH [12] is another tool using function summaries. However, it relies on quantifier elimination to compute the summaries, which is expensive and performs parallel computation of may and must summaries which increases complexity. SATURN [1, 11] has been applied successfully to find bugs in the LINUX kernel. It is summary-based but bottom-up, and the sizes of summaries are bounded in order to ensure termination. An extension of SATURN, CALYSTO [3] can extract counter-examples.

Finally, the intra-procedural trace refinement approach of [13] that we build on is implemented in ULTIMATE Automizer [20]. It has been extended to inter-procedural programs in [14]. The extension is very elegant and uses nested words to model inter-procedural traces and the corresponding notion of nested interpolant automata to prove partial correctness of recursive programs. However, it requires trace in-lining and thus is not modular. Designing a fully modular approach based on trace refinement is thus a challenge and the method we propose in this paper is a non obvious extension.

8 Conclusion

We have proposed a new algorithm which performs inter-procedural analysis in a fully modular way. Our algorithm extends the intra-procedural trace refinement algorithm of [13]. It analyses a function using available summaries for other functions and never performs any form of function or trace in-lining; it refines a function’s summary each time the function is analysed; it is top-down, context-sensitive and provides a counter-example when a program is incorrect. We have implemented the algorithm in a prototype analysis tool. We have analysed small non-recursive programs inspired from industrial case studies that contain inter-procedural defects or are hard to prove correct. The results are promising, and the next step is to demonstrate that the approach is scalable, which we believe is the case. Indeed, we can easily obtain a parallel version of our algorithm, as checking whether each function call (line 8 in Algorithm 3) satisfies a pre/postcondition can be done concurrently. Our method is also robust and independent of the technique to establish the validity of Hoare triples. Any suitable analysis technique e.g., abstract interpretation, bounded model-checking, etc. can be used.

Finally, the method we presented also suggests we break existing functions into smaller parts (almost out-lining), e.g., sequences, while loops, if statements. This makes units to analyse smaller, increases independence and enables us to compute valid Hoare triples that are often re-usable like loop invariants.

Acknowledgments. The authors would like to thank the anonymous referees for their most valuable comments that helped in preparing the final version of this paper.

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

Summary-Based Inter-Procedural Analysis via Modular Trace Refinement


