Summary-Based Inter-Procedural Analysis via Modular Trace Refinement

Franck Cassez[1], Christian Müller[2], and Karla Burnett[1]

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 = 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`).

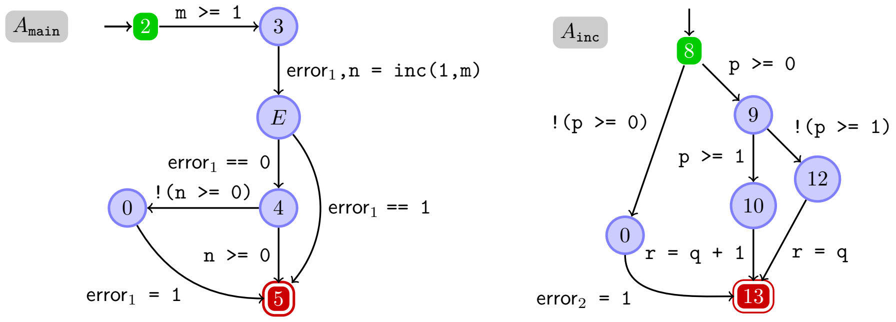

Figure 1 Function automata for program P_1.

Functions can be annotated with `assume` statements (e. g., `m >= 1` holds after line 2) and `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 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 `assert` statement violations across function boundaries in a modular manner, we introduce an extra[1] *output* variable, error, together with additional edges in the CFG of a function: *i)* a call like `n = inc(1,m)` is virtually expanded in `(error, n) = inc(1,m)` i. e., each function returns its internal error status; *ii)* each statement `assert(`φ`)` is viewed as an `if` statement, such that if φ does not hold the variable 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 $A_{\mathtt{inc}}$; *iii)* as errors may occur during function calls, the status of the caller's 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 $A_{\mathtt{main}}$.

Program P_1 has two possible causes of error: one is to call `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 `main`. The (partial) correctness of `inc` can be expressed by the Hoare triple[2] $\{\neg\mathsf{error}_2\}$ `inc` $\{\neg\mathsf{error}_2\}$. To check whether the Hoare triple $\{\neg\mathsf{error}_2\}$ `inc` $\{\neg\mathsf{error}_2\}$ 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 $A_{\mathtt{inc}}$ as a language acceptor with initial state *entry* (green) and final state *exit* (red) and the existence of a counter-example amounts to finding a *feasible* trace in $\mathcal{L}(A_{\mathtt{inc}})$, the language accepted by $A_{\mathtt{inc}}$. Determining whether such a feasible trace exists can be achieved using an iterative trace refinement algorithm [13, 15]: 1) Pick a trace[3] $w \in \mathcal{L}(A_{\mathtt{inc}})$; 2) Add the pre/post conditions $\neg\mathsf{error}_2/\mathsf{error}_2$ to the trace w and check whether

```
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  }
```

Listing 1 Program P_1

[1] We introduce one indexed error_k variable per function to simplify the technical developments. We can equivalently use a global error variable to track the error status of a complete program.
[2] We use error (resp. ¬error) as a shorthand for "every valuation such that 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 $w' = \neg\mathsf{error}_2 \cdot w \cdot \mathsf{error}_2$ 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 $\mathcal{L}(A_{\texttt{inc}}) \setminus \{w\}$. If $\mathcal{L}(A_{\texttt{inc}}) \setminus \{w\}$ is empty, the triple is valid; otherwise we start again at step 1 using a *refined* language $\mathcal{L}(A_{\texttt{inc}}) \setminus \{w\}$ which does not contain w. Notice that this process may not terminate[4]. 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 $\{P\}$ `f` $\{Q\}$ holds, we get a *better* triple $\{P'\}$ `f` $\{Q'\}$ with $P \implies P'$ (weaker assumption on input) and $Q' \implies Q$ (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[5] `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 `inc` could be $\mathtt{p} \geq 1 \implies \mathtt{r} \leq \mathtt{q}+1$ or, if we do not know anything about `inc`, $True \implies True$, which means that the output variables can be assigned any value. To determine the status of triple (H) $\{\neg\mathsf{error}_1\}$ `main` $\{\neg\mathsf{error}_1\}$, we try to find a witness trace in `main` invalidating it, i. e., starting in $\neg\mathsf{error}_1$ and ending in error_1:

1. let $w_1 =$ `m >= 1` $\cdot$ error_1`,n = inc(1,m)` $\cdot$ error_1 be a trace in $\mathcal{L}(A_{\texttt{main}})$.
2. Using the semantics of each statement, and the over-approximate summary semantics $(True, True)$ for the function call, check whether $w_1' = \neg\mathsf{error}_1 \cdot w_1 \cdot \mathsf{error}_1$ is feasible. It is and we get a witness *assignment* for the values of the variables in w_1' that implies a pre/postcondition $\pi_1 : \mathtt{p} = 1 \wedge \mathtt{q} = 1 \wedge \neg\mathsf{error}_2/\mathsf{error}_2$ for `inc` to make w_1' feasible.
3. To determine whether π_1 can be satisfied, we can establish the status of the *opposite* triple $\{\mathtt{p} = 1 \wedge \mathtt{q} = 1 \wedge \neg\mathsf{error}_2\}$ `inc` $\{\neg\mathsf{error}_2\}$.
   This triple *holds* and thus the corresponding witness pre/postcondition π_1 in `main` is infeasible. While establishing this, we have computed a *stronger valid* triple, $\{\mathtt{p} \geq 0 \wedge \neg\mathsf{error}_2\}$ `inc` $\{\neg\mathsf{error}_2\}$ that can, from now on, be used as a valid summary $(G_1, S_1) = (\mathtt{p} \geq 0 \wedge \neg\mathsf{error}_2, \neg\mathsf{error}_2)$ for `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 >= 1` $\cdot$ error_1`,n = inc(1,m)` $\cdot$ $\neg\mathsf{error}_1$ $\cdot$ `!(n >= 0)` $\cdot$ $\mathsf{error}_1 = 1$ and check whether the extended trace $w_2' = \neg\mathsf{error}_1 \cdot w_2 \cdot \mathsf{error}_1$ is feasible. It is *provided* the call to `inc` can realise an input/output constraint given by a witness assignment for w_2'. Function `inc` cannot realise this witness assignment and the result of this check is a new valid triple for `inc`, $(G_2, S_2) = (\mathtt{p} \geq 1 \wedge \mathtt{q} \geq 1, \mathtt{r} \geq \mathtt{q}+1)$.
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 `main`.
7. There is only one trace left to explore in `main` but it cannot set error_1. The final result is a triple $\{\neg\mathsf{error}_1\}$ `main` $\{\neg\mathsf{error}_1 \wedge \mathtt{n} \geq \mathtt{m}+1\}$ 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 φ, $\textsc{Var}(\varphi)$ is the set of variables appearing in φ. 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 $P \wedge Q$ (logical and) or $P \cap Q$ (set intersection). False corresponds to the empty set and True to the set of all possible values.

The set of program *statements* Σ 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 `r1,···,rk = f(d1,···,dn)` where `f` is a function and `r1,···,rk` and `d1,···,dn` are the input and output variables.

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

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

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

A function `f` is represented by a function automaton which is a trace automaton $A_{\mathtt{f}} = (\mathrm{L}_{\mathtt{f}}, \delta_{\mathtt{f}}, \{\mathit{init}_{\mathtt{f}}\}, \{\mathit{exit}_{\mathtt{f}}\})$. It is obtained from the CFG of `f` by adding the edges setting the error variable to encode `assert` statement violations (see $A_{\mathtt{main}}$ and $A_{\mathtt{inc}}$ 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 $\forall t \in \mathcal{L}(A), \mathsf{post}(t, P) \subseteq Q$. Program (or function) correctness [13] is defined by: $\{P\}$ `f` $\{Q\}$ is valid iff $\{P\}\ A_{\mathtt{f}}\ \{Q\}$ is valid. The validity of a Hoare triple $\{P\}\ A_{\mathtt{f}}\ \{Q\}$ can be expressed in terms of language inclusion:

▶ **Theorem 1.** *$\{P\}\ A_{\mathtt{f}}\ \{Q\}$ is valid iff $\mathcal{L}(A_{\mathtt{f}}) \subseteq \mathsf{Infeas}(P, \neg Q)$.*

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 $I_0, I_1, \cdots, I_k$ is a (P, Q)*-interpolant* for t if: 1) $P \implies I_0$, 2) $\forall 1 \leq i \leq k, \mathsf{post}(st_i, I_{i-1}) \subseteq I_i$ and 3) $I_k \wedge Q = \mathit{False}$. For $t \in \mathsf{Infeas}(P, Q)$, we let $\mathsf{itp}_{P,Q}(t)$ be the set of (P, Q)-interpolants for t. For $t \in \mathsf{Infeas}$, we let $\mathsf{itp}(t)$ be the set of interpolants for t. By Craig's interpolation theorem [10], we know that $\mathsf{itp}(t) \neq \varnothing$. It follows that:

▶ **Lemma 2.** *If* $t \in \mathsf{Infeas}(P, Q)$ *then* $\mathsf{itp}_{P,Q}(t) \neq \varnothing$.

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 $\mathcal{I} = I_0, I_1, \cdots, I_k$ be an interpolant for t. The *canonical interpolant automaton* [13] for $(t, \mathcal{I})$ is a trace automaton $A^t_{\mathcal{I}} = (\mathrm{L}_{\mathcal{I}}, \delta_{\mathcal{I}}, \{init_{\mathcal{I}}\}, \{exit_{\mathcal{I}}\})$. An important property of canonical interpolant automata is that they accept sets of infeasible traces:

▶ **Theorem 3** ([13]). *If* $t \in \mathsf{Infeas}$ *and* $\mathcal{I} \in \mathsf{itp}(t)$ *then* $\mathcal{L}(A^t_{\mathcal{I}}) \subseteq \mathsf{Infeas}$.

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

▶ **Theorem 4.** *If* $t \in \mathsf{Infeas}(P, Q)$ *and* $\mathcal{I} \in \mathsf{itp}_{P,Q}(t)$ *then* $\mathcal{L}(A(P, Q)^t_{\mathcal{I}}) \subseteq \mathsf{Infeas}(P, Q)$.

We can now introduce Algorithm 1, *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 $A_{\mathtt{f}}$ 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 $\{P\}\ A_{\mathtt{f}}\ \{Q\}$ 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 $A_{\mathtt{f}}$ have been declared $(P, \neg Q)$-infeasible and hence $\{P\}\ A_{\mathtt{f}}\ \{Q\}$ 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 $\{P'\}\ \mathtt{f}\ \{Q'\}$ is *stronger* than $\{P\}\ \mathtt{f}\ \{Q\}$ 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_i *cover* the set of traces of $A_{\mathtt{f}}$, and $A_{\mathtt{f}}$ satisfies the triple $\{\cap^n_{i=1} P_i\}\ A_{\mathtt{f}}\ \{\cup^n_{i=1} Q_i\}$.

▶ **Theorem 5.** *If Algorithm 1 terminates and returns* $\mathsf{itp}(P', Q')$ *then* $\{P'\}\ A_{\mathtt{f}}\ \{Q'\}$ *is valid and stronger than* $\{P\}\ A_{\mathtt{f}}\ \{Q\}$.

If Algorithm 1 terminates it either returns: *i*) $\mathsf{path}(t)$, and then $\{P\}\ \mathtt{f}\ \{Q\}$ does not hold and t is such that $\mathsf{post}(t, P) \cap \neg Q \not\subseteq False$, or *ii*) $\mathsf{itp}(P', Q')$ then $\{P'\}\ \mathtt{f}\ \{Q'\}$ 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 $\mathsf{post}(t, P) \cap \neg Q \not\subseteq False$. As post is exact for statements which are not function calls, t is a counter-example. For *ii*), if *Hoare*$_1$ returns $\mathsf{itp}(P', Q')$, 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: $Hoare_1(A_{\mathtt{f}}, P, Q)$

```
Input  : A function automaton $A_{\mathtt{f}}$, two predicates $P$ and $Q$.
Result : $\mathsf{itp}(P', Q')$ with $P \implies P'$ and $Q' \implies Q$ if $\{P\}\ A_{\mathtt{f}}\ \{Q\}$ holds,
         $\mathsf{path}(t)$ with $\mathsf{post}(P, t) \cap \neg Q \not\subseteq False$ otherwise.
Var    : $\overline{A}$: arrays of interpolant automata, initially empty
         $\overline{P}, \overline{Q}$: arrays of predicates, initially empty
         $n$ : integer, initially 0
1  while $\mathcal{L}(A_{\mathtt{f}}) \not\subseteq \cup_{i=1}^{n} \mathcal{L}(A_i)$ do
       /* There is a trace $t$ from entry to exit in $\mathcal{L}(A_{\mathtt{f}}) \setminus \cup_{i=1}^{n} \mathcal{L}(A_i)$ */
2      Let $t = st_1 \cdots st_k \in \mathcal{L}(A_{\mathtt{f}}) \setminus \cup_{i=1}^{n} \mathcal{L}(A_i)$;
3      if $\mathsf{post}(t, P) \cap \neg Q \not\subseteq False$ then
           /* $\mathsf{post}(t, P) \subseteq Q$ does not hold, $t$ is a counter-example */
4          return $\mathsf{path}(t)$;
5      else
           /* $t$ is $(P, \neg Q)$-infeasible, we refine and iterate */
6          Let $\mathcal{I} = I_0, \cdots, I_k \in \mathsf{itp}_{P,\neg Q}(t)$;
7          Let $n := n + 1$;
8          Let $A_n = A(P, \neg Q)^{t}_{\mathcal{I}}$;
9          let $P_n := I_0$ and $Q_n := I_k$;
10 return $\mathsf{itp}(\cap_{i=1}^{n} P_i, \cup_{i=1}^{n} Q_i)$;
```

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 $Hoare_1(\mathtt{f}, P, Q)$ 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 $\mathtt{f}$ be a function with (formal) input parameters $\overline{x} = x_1, \ldots, x_k$ and output parameters $\overline{y} = y_1, \cdots, y_n$. We assume that input variables are not modified by a function and there are no global variables[6]. A *summary* $S(\overline{x}, \overline{y})$ for $\mathtt{f}$ is a set of pairs of predicates $(P_i(\overline{x}), Q_i(\overline{x}, \overline{y}))_{1 \leq i \leq n}$ where $\varphi(\overline{z})$ indicates that $\textsc{Var}(\varphi) = \{z_1, \cdots, z_n\}$. A summary can equivalently be viewed as a predicate $\wedge_{i=1}^{n} \big(P_i(\overline{x}) \implies Q_i(\overline{x}, \overline{y})\big)$.

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 $\mathtt{f}$ in the program, $\mathcal{C}(\mathtt{f})$ is a summary for

[6] These limitations are not compulsory but are commonly admitted [2, 18].

f. Given a context $\mathcal{C}$ we define an associated *summary post operator* $\widehat{\mathsf{post}}$ as follows:

$$\widehat{\mathsf{post}}(\mathcal{C}, st, \varphi) = \begin{cases} \mathsf{post}(st, \varphi) \text{ if } st \text{ is not a function call, or} \\ \exists \mathtt{r}.\varphi \wedge \mathcal{C}(\mathtt{f})(\mathtt{m}, \mathtt{r}) \text{ if } st \text{ is the function call } \mathtt{r = f(m)}. \end{cases}$$

In other words, only function call post operators are computed using summaries while other statements' strongest postcondition operators are preserved. As a function call `r = f(m)` only alters the output variables r, the *projection* of the predicate φ on the other variables, $\exists \mathtt{r}.\varphi$, remains true after the execution of `r = f(m)`. Moreover the target result variable r should satisfy the constraints between the formal input and output variables $\mathcal{C}(\mathtt{f})(\mathtt{m}, \mathtt{r})$ of f. $\mathcal{C}$ is an *over-approximating context* (in short over-approximation) if for every function automaton $A_\mathtt{f}$, every trace $t \in \mathcal{L}(A_\mathtt{f})$ and every predicate φ, $\mathsf{post}(t, \varphi) \subseteq \widehat{\mathsf{post}}(\mathcal{C}, t, \varphi)$. The definitions we introduced so far are also valid for the $\widehat{\mathsf{post}}$ operator: a trace t is (P, Q)-infeasible in context $\mathcal{C}$ if $\widehat{\mathsf{post}}(\mathcal{C}, t, P) \cap Q \subseteq False$, otherwise it is feasible (in this context). We write t is $\mathcal{C}$-(P, Q)-feasible (resp. infeasible) for t is (P, Q)-feasible (resp. infeasible) in context $\mathcal{C}$ If a context is an over-approximation, infeasibility in $\mathcal{C}$ implies infeasibility with the *exact* strongest postcondition post operators for the functions called. However, a trace may be (P, Q)-feasible in $\mathcal{C}$, but not (P, Q)-feasible with the *exact* post operator for each function. Valid Hoare triples can be used as over-approximations for functions:

▶ **Proposition 6.** *Let (P, Q) be two predicates on the input and input/output variables of* f *such that $\{P\}\ A_\mathtt{f}\ \{Q\}$ holds. Then $\mathsf{post}(\mathtt{r} = \mathtt{f}(\mathtt{m}), \varphi) \subseteq \widehat{\mathsf{post}}(\mathcal{C} : \mathtt{f} \mapsto \{(P, Q)\}, \mathtt{r} = \mathtt{f}(\mathtt{m}), \varphi)$.*

Proposition 6 generalises to summaries that are sets of pairs of predicates:

▶ **Theorem 7.** *Let $\mathcal{C}$ be an over-approximation,* f *a function and (P, Q) be two predicates on the input and input/output variables of* f *such that $\{P\}\ A_\mathtt{f}\ \{Q\}$ holds. The context $\mathcal{C}'$ defined by $\mathcal{C}'(h) = \mathcal{C}(h)$ for $h \neq \mathtt{f}$ and $\mathcal{C}'(\mathtt{f}) = \mathcal{C}(\mathtt{f}) \cup \{(P, Q)\}$ 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 *Status* at line 3). This step is more involved and is detailed in Algorithm 3. Algorithm 2 determines the status of a candidate triple $\{P\}\ A_\mathtt{f}\ \{Q\}$ and either returns an *inter-procedural (counter-example) path* or a *stronger* triple $\{P'\}\ A_\mathtt{f}\ \{Q'\}$. An *inter-procedural path* for a trace t of function g is inductively defined by a mapping path such that:

- for statements st in t that are not function calls, $\mathsf{path}(st) = st$,
- for st a function call `r = f(m)`, $\mathsf{path}(st)$ is an inter-procedural path for f.

n Algorithm 2, we assume that the context variable $\mathcal{C}$ is a *global variable* and is initialised with default summaries e. g., $(True, True)$ for each function. In Algorithm 2, line 3, the call to $Status(t, P, \neg Q)$ (*Status* is defined in Algorithm 3) returns the status of t: it is either $(P, \neg Q)$-feasible and $(True, \mathsf{path}(t))$ is returned or $\mathcal{C}$-$(P, \neg Q)$-infeasible and $(False, \bot)$ is returned ($\bot$ stands for the void path). *Hoare*$_2$ is very similar to Algorithm 1 once the status of a trace t is determined:

- if t is $\mathcal{C}$-$(P, \neg Q)$-infeasible i. e., $Status(t, P, \neg Q) = (False, \bot)$, then it is $(P, \neg Q)$-infeasible ($\mathcal{C}$ 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., $Status(t, P, \neg Q) = (True, \mathsf{path}(t))$ and t is an inter-procedural counter-example.

```
Algorithm 2: $Hoare_2(A_{\mathtt{f}}, P, Q)$
  Global: $\mathcal{C}$: context with summaries for each function, initially $(True, True)$
  Input : A function automaton $A_{\mathtt{f}}$, two predicates $P$ and $Q$.
  Result : If $\{P\}\ A_{\mathtt{f}}\ \{Q\}$ holds, $\mathsf{itp}(P', Q')$ with $P \implies P'$ and $Q' \implies Q$
           Otherwise an inter-procedural path $\mathsf{path}(t)$ with $\mathsf{post}(P, t) \cap \neg Q \not\subseteq False$.
  Local : $\overline{A}$: arrays of interpolant automata, initially empty
          $\overline{P}, \overline{Q}$: arrays of predicates, initially empty
          $n$: integer, initially 0
  /* Main refinement loop */
1 while $\mathcal{L}(A_{\mathtt{f}}) \not\subseteq \cup_{i=1}^{n}\mathcal{L}(A_i)$ do
      /* There is a trace from entry to exit in $\mathcal{L}(A_{\mathtt{f}}) \setminus \cup_{i=1}^{n}\mathcal{L}(A_i)$ */
2     Let $t = st_1 \cdots st_k \in \mathcal{L}(A_{\mathtt{f}}) \setminus \cup_{i=1}^{n}\mathcal{L}(A_i)$;
      /* Determine the status of $t$ (and update summaries $\mathcal{C}$) */
3     $R, \mathsf{path}(t) := Status(t, P, \neg Q)$;
      /* Status of $t$ is settled */
4     if $R$ then
          /* $\{P\}\ A_{\mathtt{f}}\ \{Q\}$ is not valid and $\mathsf{path}(t)$ a counter-example */
5         return $\mathsf{path}(t)$;
6     else
          /* $t$ is $(P, \neg Q)$-infeasible, we refine and iterate */
7         Let $\mathcal{I} = I_0, \cdots, I_k \in \mathsf{itp}_{P,\neg Q}(t)$;
8         $n := n + 1$;
9         $A_n := A(P, \neg Q)^{t}_{\mathcal{I}}$;
10        $P_n := I_0$ and $Q_n := I_k$;
  /* $\{P\}\ A_{\mathtt{f}}\ \{Q\}$ is valid.  Add to $\mathcal{C}$ and returns a stronger summary */
11 $\mathcal{C}(\mathtt{f}) := \mathcal{C}(\mathtt{f}) \cup \{(\cap_{i=1}^{n} P_i, \cup_{i=1}^{n} Q_i))\}$;
12 return $\mathsf{itp}(\cap_{i=1}^{n} P_i, \cup_{i=1}^{n} Q_i)$;
```

$Status(t, P, \neg Q)$ 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 $t = st_1 \cdot st_2\ \cdots\ st_k$ as follows:

- function call statements are collected and stored into FCall (line 2). Then 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 $\mathcal{C}$-$(P, \neg Q)$-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 $\mathcal{C}$ is an over-approximation, and we can return $(False, \bot)$.
  - if t is $\mathcal{C}$-$(P, \neg Q)$-feasible, we obtain some before/after witness values for the variables for each function call and store them in pairs $(\nu_i, \mu_i), i \in$ 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 $\mathsf{path}(u)$ or a pair of predicates $\mathsf{itp}(P', Q')$. If we get a witness trace we store it in $\mathsf{path}(st_i)$ (line 12), otherwise we do nothing (but the context $\mathcal{C}$ has been updated by the call to $Hoare_2$).

▶ 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: $Status(t, P, \neg Q)$

```
Global : $\mathcal{C}$: context with summaries for each function
Input  : A trace $t$, two predicates $P$ and $\neg Q$
Result : $(False, \bot)$ if $t$ is $\mathcal{C}$-$(P, \neg Q)$-infeasible and thus $(P, \neg Q)$-infeasible
         $(True, \mathsf{path}(t))$ with $\mathsf{path}(t)$ a $(P, \neg Q)$-feasible full inter-procedural path.

1  Let $t = st_1 st_2 \cdots st_k$;
2  Let $\mathsf{FCall} = \{1 \le i \le k \mid st_i$ is a function call$\}$;
   /* Initialise path($t$) for regular statements */
3  foreach $i \in \{1, \cdots, k\} \setminus \mathsf{FCall}$ do $\mathsf{path}(st_i) := st_i$;
4  while $True$ do
5      if $\widehat{\mathsf{post}}(\mathcal{C}, t, P) \cap \neg Q \not\subseteq False$ then
           /* $t$ is $(P, \neg Q)$-feasible under $\mathcal{C}$ */
6          Let $\{(\nu_i, \mu_i), i \in \mathsf{FCall}\}$ be the set of witness before/after values;
           /* Check whether each function call step is feasible */
7          foreach $i \in \mathsf{FCall}$ do $\mathsf{path}(st_i) = \bot$;
8          foreach $i \in \mathsf{FCall}(t)$ do
9              Let $st_i$ be a call to f with f defined by f($\overline{x}$):$\overline{y}$ ;
10             switch $Hoare_2(\overline{x} = \nu_i, A_f, \neg(\overline{y} = \mu_i))$ do
11                 case path($u$)
                       /* $u$ s.t. post$(u, \overline{x} = \nu_i) = \overline{y} = \mu_i$ */
12                     $\mathsf{path}(st_i) := \mathsf{path}(u)$;
13                 case itp($P'$, $Q'$)
                       /* f satisfies $\{\overline{x} = \nu_i\}$ $A_q$ $\{\neg(\overline{y} = \mu_i)\}$ */
                       /* $(P', Q')$ has been added to summary of f */
14         if $\bigwedge_{l \in \mathsf{FCall}} \mathsf{path}(st_l) \neq \bot$ then
15             return $(True, \mathsf{path}(t))$
16     else
           /* $t$ is $(P, \neg Q)$-infeasible under $\mathcal{C}$ and thus $(P, \neg Q)$-infeasible */
17         return $(False, \bot)$;
```

a location associated with an interpolant I to another associated with J if $\mathsf{post}(st, I) \subseteq J$. As the contexts contain only over-approximations for function calls we can safely check whether a back edge can be added or not. Checking whether $\mathsf{post}(st, I) \subseteq J$ still requires an SMT-solver even if we use inductive interpolants computed using weakest preconditions.

The following Theorem establishes the (partial) correctness of Algorithm 2:

▶ **Theorem 8.** *Let $\mathcal{C}$ be an initial over-approximation. If $Hoare_2(A_\mathtt{f}, P, Q)$ terminates and there are less than n calls to $Hoare_2$, then:*

a) *the result of $Hoare_2(A_\mathtt{f}, P, Q)$ is correct i. e.,*
  1. *if it returns* $\mathsf{itp}(P', Q')$, *$\{P'\}$ $A_\mathtt{f}$ $\{Q'\}$ holds, with $P \implies P'$, $Q' \implies Q$,*
  2. *if it returns* $\mathsf{path}(t)$ *then* $\mathsf{path}(t)$ *is a finite inter-procedural path and* $\mathsf{post}(P, \mathsf{path}(t)) \cap \neg Q \not\subseteq False$,

b) *during the computation $\mathcal{C}$ is always an over-approximation.*

Theorem 8 proves that $Hoare_2$ is sound by $a.1$). If the Hoare triple is not valid, and if the post operator is exact then the returned counter-example is also feasible by $a.2$). The algorithm is also trivially complete (as in [13]) relative to the existence of a modular Hoare proof for the non-recursive program: if a program is error-free, there exists a context $\mathcal{C}$ such that we can establish correctness.

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

1 Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In Manuvir Das and Dan Grossman, editors, *PASTE*, pages 43–48. ACM, 2007.

2 Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *LNCS*, pages 39–55. Springer, 2012.

3 Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 211–220. ACM, 2008.

4 Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

5 Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

6 Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.

7 Sebastian Biallas, Mads Chr. Olesen, Franck Cassez, and Ralf Huuck. Ptrtracker: Pragmatic pointer analysis. In *SCAM*, pages 69–73. IEEE, 2013.

8 Mark Bradley, Franck Cassez, Ansgar Fehnker, Thomas Given-Wilson, and Ralf Huuck. High performance static analysis for industry. *ENTCS*, 289:3–14, 2012.

9 Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An Interpolating SMT Solver. In Alastair F. Donaldson and David Parker, editors, *SPIN*, volume 7385 of *LNCS*, pages 248–254. Springer, 2012.

10 William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. Symb. Log.*, 22(3):269–285, 1957.

11 Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*, pages 270–280. ACM, 2008.

12 Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In Hermenegildo and Palsberg [16], pages 43–56.

13 Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In Jens Palsberg and Zhendong Su, editors, *SAS*, volume 5673 of *LNCS*, pages 69–85. Springer, 2009.

14 Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In Hermenegildo and Palsberg [16], pages 471–482.

15 Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 36–52. Springer, 2013.

16 Manuel V. Hermenegildo and Jens Palsberg, editors. *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM, 2010.

17 Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995.

18 Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-based function summaries in bounded model checking. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *HVC*, volume 7261 of *LNCS*, pages 160–175. Springer, 2011.

19 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. Technical report, 1978. Dpt. Of Computer Science, Courant Institute, NY, USA.

20 Ultimate Automizer. `http://ultimate.informatik.uni-freiburg.de/automizer/`.