Scaling Automatic Modular Verification

Lauren Pick

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Aarti Gupta

January 2022

Abstract

Automated software verification techniques, while widely successful, often suffer from scalability issues due to state-space explosion. In both automated and manual verification, modular approaches help scale verification by breaking verification problems into several easier-to-solve subproblems. These subproblems often involve discovering suitable invariants that can be used to help derive a proof that the entire system meets the desired specification.

In this dissertation, I describe work toward developing modular automatic techniques for software verification in which such invariants are discovered automatically. These techniques notably involve exploiting the structure and syntax of both system components and/or their specifications in order to find useful invariants for scaling verification. I have developed techniques for several related kinds of verification problems: the verification of k-safety properties, the verification of safety properties for general single-threaded interprocedural programs, and the verification of information-flow security–a specific kind of 2-safety property.

For each of these verification problems, I have implemented the developed techniques in a verification tool and compared the tool to existing state-of-the-art tools for solving the verification problem. Experimental results demonstrate that the developed techniques help scale verification over existing state-of-the-art and allow more verification problems to be solved automatically.

Acknowledgements

I would like to thank my advisor Aarti Gupta for her guidance and encouragement. I am immensely grateful to have had such an exceptionally supportive advisor throughout the years of my Ph.D.

I would also like to thank the members of my thesis committee: Grigory Fedyukovich, Andrew Appel, Lennart Beringer, and Zak Kincaid. I would additionally like to thank David Naumann and Pamela Zave, who along with my thesis committee have provided useful feedback on the content of my thesis. I would especially like to thank my readers Grigory Fedyukovich and Andrew Appel for their detailed comments on the thesis itself.

I am grateful to those who helped me cultivate and explore my interest in formal methods and programming languages as an undergraduate. In particular, I would like to thank Alwyn Goodloe for the opportunity to intern at NASA Langley as an undergraduate, which was instrumental for encouraging my early interest in formal methods and programming languages. I would also like to thank Alan Mycroft, Ali Sezgin, and Dominic Mulligan for their help in supervising my undergraduate disseration and encouragement.

I would like to thank for hosting me as an intern at SRI International. I would also like to thank Daniel Schwartz-Narbonne and Ankush Desai for hosting my internships at Amazon Web Services. Each internship has been a very different experience and has enriched my experience as a researcher in distinct ways.

I am deeply grateful for the Princeton Programming Languages group for providing a fun, comfortable, and vaguely bubble-tea-flavored environment throughout my time at Princeton.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656466. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The work included in this dissertation was also supported in part by NSF Grants 1525936, 1837030, and 2107138.

Finally, I would like to thank my family and especially my parents for all their support.

Contents

Chapter 1

Introduction

There are a great many techniques and approaches for ensuring the correctness of software systems, including those that fall within the categories of testing and formal verification. Formal verification can be used to solve the *software verification problem* of determining that a given program meets a given specification under all inputs. In contrast, testing cannot in general provide guarantees about all possible inputs to the system.

Formal verification provides guarantees of correctness by relying on finding proofs that a system meets a given specification. Over the years, there have been many efforts to apply formal verification to software systems. The techniques used in these efforts vary greatly in the level of automation involved.

On the less automated side, users can operate proof assistants such as Coq [2] and Agda [1] to help in the manual construction of a machine-checkable proof. While some automation can be achieved through the use of tactics within a proof script or through proof search, ultimately the user must decide when to apply the automation and must contribute substantial manual effort. On the automated side, software model checking techniques [98] employ algorithms that, with no user input, prove that a provided program meets a provided specification. Successful automated verification techniques have relied on backend Satisfiability Modulo Theories (SMT) solvers [24] with great success, encoding entire verification problems or their subproblems as SMT formulas that can be solved by powerful backend solvers such as Z3 [61] or CVC4 [23]. Even certain proof assistants like F* [147] have made extensive and successful use of SMT solvers. Of course, fully automated efforts may fail because the software verification problem is in general undecidable and, unlike in approaches in which the user is more involved, there is a lack of human insight that could help for a particular program and property.

Significant challenges in applying automated verification result from this lack of human insight. One challenge in particular lies in the lack of information about how to perform verification *modularly*, i.e., how to reduce the solving of a large verification problem involving a complex piece of code into solving smaller verification subproblems. Automated verification techniques, which often rely on backend SMT solvers [24], require such a reduction in order to scale verification to larger programs. A simple monolithic encoding of a verification problem as a single SMT solver query is unlikely to yield a result from an SMT solver in a timely manner, if at all.

For automated tools that incorporate human insights, the generation of verification subproblems can be performed under user guidance. For example, many tools allow or even require users to provide annotations that express *program invariants* [107], which are additional properties about the program that the automated technique may not be able to infer by itself. Ideally, these invariants should be able to be both easily proved by the automated verifier and easily used by the verifier to prove that the given specification holds. In other words, these invariants should allow the verifier to solve two separate, smaller problems: (1) showing that the invariants hold and (2) showing that the specification for the program holds given the invariants. While using human guidance is helpful for a modular verification approach and has led to the successful application of verification to real-world software systems [87, 88], invariants that are both useful and easy to prove can be difficult for a user to generate. In fully automated techniques that do not use such annotations, the automatic inference of such invariants can present a significant challenge, but, if successful, can alleviate the burden on the user to manually construct such invariants.

One successful approach for invariant inference has been based on the use of syntax-guided synthesis (SyGuS) [8]. Given a grammar and a specification of a property that a desired derivation of the grammar should satisfy, the problem of syntax-guided synthesis is to find such a term that satisfies the specification. SyGuS has been applied successfully to program synthesis for particular domains [8], where derivations of the grammar are programs or program expressions and the specification is that the program has a particular property, and to the synthesis of certain kinds of invariants [72, 73, 74, 33, 154, 138, 78, 120, 10, 129], where terms of the grammar are logical formulas and the specification is that the generated formula is, indeed, a program invariant.

SyGuS-based techniques, while successfully applied to particular domains, also face scalability limitations. As the number of terms in the grammar grows, so does the search space that a SyGuS solver must explore. While smarter search heuristics [72, 73, 138, 129] and other techniques for pruning the search space exist to help scale SyGuS techniques, the success of SyGuS still relies heavily on the careful design of grammars that restrict the search space sufficiently that it can be explored but still are expressive enough to contain the desired solutions to SyGuS problems. In general, these techniques require human insight in the design of grammar templates, but they require less human effort than would be required for creating the solution manually. Furthermore, they can reuse the human effort (i.e., the grammar) for other problems in the same domain; for a SyGuS-based synthesizer, the main human insight required is the up-front identification of relevant pieces of syntax.

In this dissertation, I present fully automated techniques for verification that leverage syntactic features of both programs and properties to perform effective automatic decomposition of a verification problem into subproblems. The development of these techniques are based on the insight that syntactic and structural features of programs and properties capture, to some extent, human insights about the verification problem being solved. In developing these techniques, I have targeted them to particular kinds of verification problems whose instances share similar syntactic features that can be leveraged. For example, one kind of verification problem I have considered involves proving that programs exhibit certain k-safety properties, which are properties about k runs of the same program.

The general approach employed in the development of these techniques involves identifying scalability issues for verifying the kinds of verification problems being considered, identifying useful syntactic or structural features of the programs and properties, and finally using these features to help decompose the verification problem into simpler subproblems. The final step in this approach involves using the syntactic and structural features of the program and property to help perform invariant inference.

1.1 Contributions

I have applied this approach of identifying and using relevant syntactic and structural features to help scale verification for three related kinds of verification problems: the verification of k-safety properties for intraprocedural programs, the verification of 1-safety properties for interprocedural programs, and the verification of information-flow properties for interprocedural programs. The resulting technical contributions of the work described in this dissertation are described below.

k-safety properties. For the verification of k-safety properties, I developed a method for decomposing corresponding loops across the k different runs into *maximal* *sets of loops* to consider in verification subproblems. Each set of loops constitutes a verification subproblem, and decomposition into maximal sets of loops allows for fewer and simpler invariants to be used in solving these verification subproblems, ultimately improving scalability.

I also contributed a novel application of *symmetry-breaking* to eliminate redundant verification subproblems when checking k-safety. Symmetries are identified using the syntax of properties in verification subproblems and are used to avoid solving unnecessary verification subproblems, leading to a clear performance improvement.

I implemented a tool SYNONYM applying these techniques in an existing automated k-safety verifier. These contributions are described in more detail in Chapter 3 and have been presented in a previous conference paper [123].

Interprocedural programs. For the verification of 1-safety properties for interprocedural programs, I proposed a parameterizable method of generating verification subproblems for interprocedural programs using the novel notion of *bounded environments*. Like other modular techniques for handling interprocedural programs [103, 82, 112], the problem of verifying the interprocedural program is broken up into subproblems that involve inferring invariants for a procedure at a time; the result of using bounded environments is that the scope of verification subproblems is controlled by a user-provided parameter and follows the structure of the program call graph. The parameterizable nature of bounded environments allows a user to adjust the size of verification subproblems by adjusting the parameter and consequently impact the scalability of verification overall.

I also proposed *EC lemmas*, a novel form of invariant useful for handling mutually recursive procedures, and a way to learn EC lemmas. These invariants are of a particular form influenced by the program call graph that is suited for the handling of mutual recursion.

Finally, I developed an algorithm that uses bounded environments and EC lemmas for verifying interprocedural programs, implemented in a tool called Clover. These contributions are described in more detail in Chapter 4 and have been presented in a previous conference paper [125].

Information-flow properties of interprocedural programs. For the verification of information-flow properties for interprocedural programs, I developed *grammar templates* that can be used in a SyGuS-based approach for inferring useful information-flow invariants with and without quantifiers. Information-flow properties can be formalized as 2-safety properties [148], so useful invariants include those that can express relationships about variables across the two runs. The developed grammar templates thus capture syntactic features such as the equalities of corresponding variables and array elements across different runs. The grammar templates also include property-directed invariants that are useful for verification in the presence of the *declassification* of otherwise high-security data.

The use of these grammar templates for invariant inference allows for completely automated verification of information-flow properties where previous techniques that took a modular approach to interprocedural program verification would have required human annotations of intermediate invariants. In other words, these grammar templates encode the necessary human insights to enable the solving of the verification subproblems generated when performing modular verification of information-flow properties for interprocedural programs. I developed and implemented an algorithm that applies these grammar templates in order to verify information-flow properties of interprocedural programs, implemented in a tool called Flower. These contributions are described in more detail in Chapter 5 and have been presented in a previous conference paper [124].

Comparisons to state-of-the-art. For each tool developed, I also provide a comparison of the performance of the tool to existing state-of-the art tools for solving the kinds of verification problems being considered. These comparisons indicate that the techniques developed are helpful for scaling automated verification for these kinds of problems, enlarging the set of programs to which automated verification can be usefully applied.

Chapter 2

Preliminaries

In this chapter, I will describe relevant background information and define useful concepts that will be used throughout the remainder of the dissertation. In particular, I will describe different approaches to performing formal reasoning over software and the forms of invariants that are used in these different approaches.

For discussing what kinds of invariants may be used and how they may be used for verifying safety properties of programs, the simple C-like program shown in Figure 2.1 will serve as an ongoing example. The verification problem here is to prove that the assertion in the `main` procedure always holds under the assumptions made in the earlier `assume` statements.

```
main(x, y) {
  assume(x > 1);
  assume(x < y);
  while (y < 10) {
    y := update(x, y);
  }
  assert(x < y);
}

update(x, y) returns z {
  z := x * y;
}
```

Figure 2.1: Example program

2.1 First-Order Logic

Many program properties can be expressed using first-order logic, including specifications and relevant intermediate properties for verification. In software model checking, first-order logic can be used to encode the entire verification problem; first-order logic formulas both model the system and encode the desired property about the system. The following grammar gives the syntax for a first-order logic formula [37], with the "true" and "false" truth constants respectively denoted by $\top$ and $\bot$:

$$\phi := R(\vec{t}) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists x.\phi \mid \top \mid \ \bot$$

Each R denotes a predicate symbol and each $\vec{t}$ denotes a vector $(t_1, \ldots, t_n)$ of *terms*, where terms are given by the following grammar:

$$t := x \mid c \mid f(\vec{t})$$

Variables are denoted by x, constants by c, and functions by f.

The logical symbols for disjunction ($\vee$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), and universal quantification ($\forall$) are not included in the grammar for ϕ but can be defined in the usual way from conjunction ($\wedge$), negation ($\neg$), and existential quantification ($\exists$). In the sequel, these additional logical symbols are used and assumed to be defined in this way.

For verification, it is often desirable to use theories, which can formalize existing knowledge about integers, arrays, and other structures commonly seen in programs. The success of SMT solvers, which can solve first-order logic formulas with theories, has also led to the success of first-order logic in program verification. A first-order theory $\mathcal{T} = (\Sigma, \mathcal{A})$ consists of a set Σ of constant, function, and predicate symbols, and a set $\mathcal{A}$ of first-order logic formulas containing only constants, functions, and predicate symbols in Σ. These axioms $\mathcal{A}$ provide *interpretations* for the functions and predicate symbols in the signature Σ. Functions and predicate symbols outside of Σ are said to be *uninterpreted*.

This dissertation makes use of both the theory of integers and the theory of arrays. The theory of integers has a signature containing integers as constant symbols; multiplication by an integer constant, addition ($+$), and subtraction ($-$) as functions; and equality ($=$) and ($>$) as predicate symbols. The theory of arrays has a signature containing select ($\cdot[\cdot]$) and store ($\cdot\langle\cdot \lhd \cdot\rangle$) functions as well as an equality ($=$) predicate symbol.

2.2 Hoare Logic

Hoare logic is a widely-used formalism for reasoning about the correctness of programs using Hoare triples and inference rules on them [92]. Proving correctness of a program is done via the application of inference rules on Hoare triples. A *Hoare triple* is a triple $\{P\}\ S\ \{Q\}$ consisting of a *precondition* P, a *program statement* S, and a *postcondition* Q. The precondition P and postcondition Q are both formulas in first-order logic. Using partial correctness semantics, a Hoare triple $\{P\}\ S\ \{Q\}$ is *valid* iff, given a state in which P holds initially, executing the program statement S leads to a state in which Q holds or else leads to a nonterminating execution. Annotations in Figure 2.2 represent the Hoare triples needed to prove the assertion in the program in Figure 2.1 holds, where each triple $\{P\}\ S\ \{Q\}$ within Figure 2.1 is a Hoare triple that could be inferred through the application of Hoare logic proof rules. Each Hoare triple must be such that its precondition $\{P\}$ is the same as the preceding Hoare triple's postcondition $\{Q\}$. Hoare logic thus typically proceeds in a forward or backward manner.

Important invariants to consider when using Hoare logic are *loop* invariants. These should hold when a loop is reached and after each iteration of the loop. For the `while` loop in Figure 2.2, this loop invariant is given by $1 < \texttt{x} < \texttt{y}$. While here the loop invariant is simply the result of the `assume` statements, in other scenarios, they may

```
main(x, y) {
  {⊤}
  assume(x > 1);
  {x > 1}
  assume(x < y);
  {0 < x < y}
  while (y < 10) {
    {1 < x < y ∧ x < 10}
    y := update(x, y);
    {1 < x < y}
  }
  {1 < x < y ∧ y ≥ 10}
  assert(x < y);
}

update(x, y) returns z {
  z := x * y;
}
```

Figure 2.2: Hoare triples for the program in Figure 2.1

not be immediately obvious and can be difficult to infer automatically. In general the strengthening of Hoare logic preconditions or weakening of postconditions may be required, and finding the necessary constraints or abstractions may similarly be difficult to do automatically.

Though inference of Hoare logic preconditions and postconditions can be difficult to automate in general, the *checking* of Hoare logic proofs has been automated successfully in tools like Dafny [107], which has been used in a variety of verification efforts [87, 88]. In Dafny [107], users provide some annotations (e.g., loop invariants and method preconditions and postconditions), and SMT solvers are used in the backend to prove that a specified property holds.

2.3 Interprocedural Analysis

Program analysis techniques provide other means to reason about programs. While Hoare logic-based approaches can, e.g., proceed in a forward- or backward-style manner into procedure calls for each separate call, this way of handling procedure calls is similar to inlining. Inlining procedure calls before performing verification or program analysis will lead to the repeated analysis of the same procedure body for any procedure called more than once. Furthermore, inlining-based approaches will not work for recursive procedures.

To tackle the issues presented by procedure calls, interprocedural program analysis techniques compute and use *procedure summaries*. A procedure summary is a kind of invariant of the behavior of a procedure invocation. Procedure summaries can be viewed as specifications or interface contracts, where internal implementation details have been abstracted away. In addition to aiding code understanding and maintenance, they can be combined to verify the full program.

After a procedure summary is inferred by the analysis, it can be used where applicable to avoid re-analyzing a procedure body in some cases. Such approaches for computing interprocedural data flow [137, 127, 18] are based on the notion of a kind of interprocedural control flow graph. This graph contains the flow graphs of all procedures but also distinguishes edges that result from intraprocedural flows from edges that result from interprocedural ones. For each edge in the graph, dataflow analyses associate a dataflow function that describes the effect of the corresponding control-flow construct on dataflow. For interprocedural flow graphs that contain edges that correspond to a function call, e.g., edges from a call block to the block immediately following the call (as used in the functional approach to interprocedural analysis [137]), the dataflow function for this edge constitutes a procedure summary.

In general, summaries for procedures either over- or under-approximate their behaviors, with over-approximate summaries capturing useful information for all program executions and under-approximate procedures capturing useful information for only some executions. While much work in program analysis computes only over- or under-approximate procedure summaries, there also exist program analyses that compute both [82]. The computation of both over- and under-approximate procedure summaries can help solve verification problems more easily [82, 102].

The separate analysis of procedure bodies and the use of the results of this analysis constitutes a *modular* approach to program analysis. The problem of performing whole-program analysis is handled modularly by handling the subproblems of performing analysis of program procedures; the results of these analyses (i.e., procedure summaries) can then be combined to solve other subproblems and ultimately the whole problem.

2.4 Constrained Horn Clauses

Many modern approaches to automated software model checking make use of an encoding of the software verification problem as a system of Constrained Horn Clauses (CHCs) [84]. For a system of CHCs encoding a verification problem, a solution to the system of CHCs exists iff the property holds for the original program. Many CHC-solving algorithms are modular and fully-automated, making use of SMT-solver backends. Combined with the ability to encode software verification problems as systems of CHCs [84], these CHC solvers can be effectively seen as modular software verifiers.

In this section, I will provide background information about CHCs, describe how to encode transition systems as systems of CHCs, and finally describe how to encode interprocedural programs as CHCs. For each encoding, I will describe the correspondence between the encoding and program invariants.

Definition 2.4.1 (CHC)**.** A CHC is an implicitly universally quantified implication, which is of the form *body* $\Rightarrow$ *head*. Let $\mathcal{R}$ be a set of uninterpreted predicates. The formula *head* may take either the form $R(\vec{y})$ for $R \in \mathcal{R}$ or else $\perp$. Implications in which *head* $=\perp$ are called *queries*. The formula *body* may take the form $\phi(\vec{x})$ or

$R_1(\vec{x}_1) \wedge \ldots \wedge R_n(\vec{x}_n) \wedge \phi(\vec{x})$, where each R_i is an uninterpreted predicate, and $\phi(\vec{x})$ is a fully interpreted formula over $\vec{x}$, which may contain all variables in each $\vec{x}_i$ and (if the head is of the form $R(\vec{y})$) all variables in $\vec{y}$.

Definition 2.4.2 (Solution)**.** A *solution* for a system of CHCs is a mapping M of predicates in $\mathscr{R}$ to interpretations, such that the interpretations for the predicates satisfy all the CHCs in the system.

Definition 2.4.3 (Inductive Interpretation)**.** For a mapping M of uninterpreted predicates to interpretations, we say that the interpretations of M are *inductive* iff they satisfy all non-query CHCs.

In particular, an M that maps each n-ary predicate R to $\lambda x_1, \ldots, x_n.\top$ is inductive. Furthermore, a solution M for a system of CHCs is also inductive.

2.4.1 CHCs for Transition Systems

As an intermediate, easier-to-understand step, we consider how to encode the safety of a transition system in a system of CHCs [84]. First we review Kripke structures and how to prove safety for them. Then we will consider their encoding as a system of CHCs.

We will consider Kripke structures (S, S_0, T, L, AP, V), where S is the set of states, $S_0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, $L : S \rightarrow 2^{AP}$ is the labeling function for states, and AP is a set of atomic propositions over state variables V. We can use the labeling function to help us refer to sets of states using formulas, where a formula $F(V)$ over V refers to states $s \in S$ such that $\bigwedge L(s) \Rightarrow F(V)$. Let $init(V)$ be a formula capturing all the states in S_0. Also let $\rho(V, V')$ be a formula over $V \cup V'$ capturing transition relation T. For any $(s_0, s_1) \in T$, $\rho(V, V') \Rightarrow L(s_0) \wedge L(s_1)'$, where V' is a set of primed variables of V and $L(s_1)'$ denotes the replacement of every variable in $L(s_1)$ with its primed version in V'.

$$\begin{aligned} \mathit{init}(V) &\Rightarrow R(V) \\ R(V) \wedge \rho(V, V') &\Rightarrow R(V') \\ R(V) \wedge \mathit{error}(V) &\Rightarrow \perp \end{aligned}$$

Figure 2.3: An encoding of the safety of a transition system as a system of CHCs

Definition 2.4.4 (Safety of transition system)**.** Given a set $\mathit{Error} \subseteq S$ of error states, a Kripke structure (S, S_0, T, L, AP, V) is *safe* iff there exists no sequence of states $s_0 s_1 \ldots s_n$ for $s_i \in S$ where $s_0 \in S_0$, $(s_i, s_{i+1}) \in T$ for $i \in \{0, \ldots n-1\}$, and $s_n \in \mathit{Error}$.

Note that for a formula $\mathit{error}(V)$ capturing the set of error states, the Kripke structure is also safe if there is no sequence of labels for states $L(s_0)L(s_1)\ldots L(s_n)$ for $s_i \in S$ where $L(s_0) \Rightarrow \mathit{init}(V)$, $L(s_i) \wedge L(s_{i+1})' \Rightarrow \rho(V, V')$ for $i \in \{0, \ldots n-1\}$, and $L(s_n) \Rightarrow \mathit{error}(V)$. We will try to show safety in this way.

One way to prove safety is to find an *invariant*. An inductive invariant $\mathit{inv}(V)$ is a formula over the variables in V such that the following hold:

$$\mathit{init}(V) \Rightarrow \mathit{inv}(V) \tag{initiation}$$

$$\mathit{inv}(V) \wedge \rho(V, V') \Rightarrow \mathit{inv}(V') \tag{consecution}$$

The initiation check demonstrates that the inductive invariant holds for all initial states of the transition system. The consecution check demonstrates that if the invariant holds for a state, then it holds for any state to which it might transition. If both checks hold and the safety check holds, then the transition system is safe, where this check is as follows:

$$\mathit{inv}(V) \wedge \mathit{error}(V) \Rightarrow \perp \tag{safety}$$

I.e., if the invariant holding in a state implies that the state is not an error state.

An encoding of the problem of proving safety as a system of CHCs (as described previously [84]) is shown in Figure 2.3. An inductive interpretation will provide an interpretation for uninterpreted predicate R that makes the first two CHCs true. Note that these correspond exactly to the initiation and consecution checks shown earlier; the interpretation for R in the inductive interpretation is an inductive invariant of the original transition system. The query CHC corresponds exactly to the final check for safety mentioned previously; there is a solution M for the system of CHCs iff there is an inductive invariant $inv(V)$ for which the safety check holds, and $inv(V) = M[R]$, where $M[R]$ denotes the interpretation to which M maps R.

2.4.2 CHCs for Interprocedural Programs

The problem of modular program verification can similarly be expressed as a system of CHCs [84].

A system of CHCs for a particular program without global variables can be generated by introducing an uninterpreted predicate per procedure and encoding the semantics of each procedure using these predicates. A procedure with n inputs and m outputs can be encoded as an $n + m$-ary predicate. An interpretation of this predicate could then express relationships between the inputs and outputs of the procedure. In the context of discussing CHC encodings, each loop is treated as a recursive procedure, assertions are encoded using queries.

For a `while` loop with condition e and body *body* that access and modify variables $\vec{v}$, the loop can be transformed into a recursive procedure by replacing the loop with a call to a new function P that has variables $\vec{v}$ as both its inputs and outputs and has `if` (e) $\{body; \vec{v} := P(\vec{v})\}$ as its body. In order for assertions to be encoded using queries, they may need to be hoisted (i.e., propagated up) from locations deeper in the call graph to the the entry procedure of a program [105].

Rather than directly encoding a version of an interprocedural data flow graph as used in program analysis, CHCs can be seen as encoding a call graph for a program.

$$
\begin{array}{ll}
\mathit{proc} & ::= P(x_1, \ldots, x_m) \ \texttt{returns} \ (y_1, \ldots, y_n) \ \{\mathit{stmt}\} \\
\mathit{stmt} & ::= \mathit{stmt}_1; \mathit{stmt}_2 \mid (x_1, \ldots, x_n) := e \mid \texttt{assume} \ e \mid \texttt{if}(e) \ \{\mathit{stmt}\} \mid \\
 & \quad (x_1, \ldots, x_n) := P(e_1, \ldots, e_m)
\end{array}
$$

Figure 2.4: The syntax of procedures

$$
\begin{array}{ll}
\mathit{Enc}(\mathit{stmt}_1; \mathit{stmt}_2) & = \mathit{Enc}(\mathit{stmt}_1) \land \mathit{Enc}(\mathit{stmt}_2) \\
\mathit{Enc}((x_1, \ldots, x_n) := e) & = (x_1, \ldots, x_n) = \mathit{Enc}(e)\} \\
\mathit{Enc}(\texttt{assume} \ e) & = e \\
\mathit{Enc}(\texttt{if}(e) \ \{\mathit{stmt}\}) & = \mathit{Enc}(e) \Rightarrow \mathit{Enc}(\mathit{smt}) \\
\mathit{Enc}((x_1, \ldots, x_n) := P(e_1, \ldots, e_m)) & = P(e_1, \ldots, e_m, x_1, \ldots, x_n)
\end{array}
$$

Figure 2.5: SMT encodings for procedure bodies

For each CHC $\mathit{body} \Rightarrow \mathit{head}$, a predicate B appears in the body where the predicate in the head is H whenever the procedure for H calls the procedure for B.

To make this more concrete, assume that there is an arbitrary program. Assume that it is transformed into a program in which each non-entry procedure has the syntax specified in Figure 2.4, with employed transformations including transforming loops into recursive procedures and performing assertion hoisting. Assume that this program is further transformed so that each procedure body is converted into single-static assignment form [55], in which each variable is assigned to exactly once, yielding a resulting program Prog. Enc in Figure 2.5 then demonstrates how to generate an SMT encoding of the body of an arbitrary procedure in Prog, where each expression e has a straightforward mapping into an SMT formula $\mathit{Enc}(e)$. Let P be an arbitrary non-entry procedure in Prog with n inputs, m outputs, and body body_P. It is possible to convert the encoding $\mathit{Enc}(\mathit{body}_P)$ of the body of P into disjunctive normal form $\mathit{DNF}(\mathit{Enc}(\mathit{body}_P))$ [37]. For each disjunct D_P in $\mathit{DNF}(\mathit{Enc}(\mathit{body}_P))$, the following CHC is generated: $D_P \Rightarrow P(x_1, \ldots, x_n, y_1, \ldots y_m)$.

```
main(x, y) {
  assume(x > 1);
  assume(x < y);
  (x', y') := while(x, y);
  assert(x' < y');
}

while(x, y) returns (x', y') {
  if (y < 10) {
    z := update(x, y);
    (x', y') := while(x, z);
  }
  if (!(y < 10)) {
    (x', y') := (x, y);
  }
}

update(x, y) returns z {
  z := x * y;
}
```

Figure 2.6: Transformed version of the program from 2.1

$$\begin{array}{ll} z = x * y & \Rightarrow \mathit{update}(x, y, z) \\ y < 10 \wedge \mathit{update}(x, y, z) \wedge \mathit{while}(x, z, x', y') & \Rightarrow \mathit{while}(x, y, x', y') \\ \neg(y < 10) \wedge x' = x \wedge y' = y & \Rightarrow \mathit{while}(x, y, x', y') \\ x > 1 \wedge y > x \wedge \mathit{while}(x, y, x', y') \wedge \neg(x' < y') & \Rightarrow \bot \end{array}$$

Figure 2.7: CHC encoding of the program from Figure 2.1

For the entry procedure with body $\mathit{body}_{\mathit{main}}$, the syntax of the procedure is as in Figure 2.4, except we additionally allow `assert` statements to appear in the procedure body, where $\mathit{Enc}(\texttt{assert}(e)) = \neg \mathit{Enc}(e)$. For each disjunct D_{main} in $\mathit{DNF}(\mathit{Enc}(\mathit{body}_{\mathit{main}}))$, the following CHC is generated: $D_{\mathit{main}} \Rightarrow \bot$.

Figure 2.6 gives the program from Figure 2.1 after all transformations have been applied. The loop has been transformed into a recursive procedure. The per-procedure conversion to single static assignment form introduced primed versions of the `x` and `y` variables. Figure 2.7 gives the final CHC encoding of the program from Figure 2.1.

Each interpretation of a predicate can be viewed as a procedure summary or specification and expresses an invariant for the procedure. These may but need not

$$\begin{aligned} \mathit{update} &\mapsto \lambda x, y, z. x > 1 \land y > 1 \Rightarrow z > y \\ \mathit{while} &\mapsto \lambda x, y, x', y'. x' = x \land y' \geq y \end{aligned}$$

Figure 2.8: Interpretations for predicates in the system of CHCs in Figure 2.7

correspond to strongest postconditions nor weakest preconditions. In the case of the example program whose encoding is shown in Figure 2.7, the interpretations shown in Figure 2.8 are sufficient for proving safety. There are other possible interpretations that also constitute solutions of the CHCs. For example, the interpretation that maps *while* to $\lambda x, y, x', y'. 1 < x < y \Rightarrow 1 < x' < y'$ would also be sufficient. In fact, this interpretation expresses a loop invariant that is the same as the loop invariant for the `while` loop shown in Figure 2.2.

For a formula F containing uninterpreted predicates, we let $M(F)$ be the result of replacing each predicate with its interpretation in M. We may regard these interpretations as procedure summaries, making this substitution analogous to replacing a procedure with its summary.

For an inductive M, for each predicate R that represents a program procedure `r`, $M[R]$ is an *over-approximation* of the behavior of procedure `r`. Each $M[R]$ expresses an *invariant* for the behavior of `r`. For example, the interpretation for *update* in Figure 2.8 captures the relationship between the inputs x, y and output z of the `update` procedure in Figure 2.1 and the interpretation for *while* captures the loop invariant for the `while` loop in Figure 2.1.

Under-approximate summaries for procedures can also be captured in interpretations for predicates. For a mapping M of predicates to interpretations that constitute under-approximate summaries, this mapping should obey the invariant that $\forall \vec{x}. M[R](\vec{x}) \Rightarrow O[R](\vec{x})$, where O is any inductive mapping of the predicates in the CHC to interpretations, including the strongest possible mapping O. Each interpretation $M[R]$ can thus be viewed as an under-approximation of the behavior of the corresponding program procedure `r` of the original program. Such mappings will be referred to *under-approximate mappings* (as opposed to *over-approximate mappings*). Under-approximate summaries can be used both to help with inferring over-approximate summaries during verification (see Chapter 4) and with finding counterexamples to correctness for buggy programs.

More specifically, for a system of CHCs, we can construct several SMT formulas such that, if any is satisfiable, the original system of CHCs has no solution. Such formula U is the body of any unfolding of a query CHC, where unfolding is defined as follows:

Definition 2.4.5 (Unfolding)**.** For a given CHC C in a system of CHCs, where C is of the form $R_1(\vec{x}_1) \wedge \ldots \wedge R_n(\vec{x}_n) \wedge \phi(\vec{x}) \Rightarrow \mathit{head}$, an uninterpreted predicate R_i in its body can be unfolded in the CHC by replacing the occurrence of $R_i(\vec{x}_i)$ with $\mathit{fresh}(\mathit{body}_i[\vec{y}_i \mapsto \vec{x}_i], \vec{x}_i, C)$, where $\mathit{body}_i \Rightarrow R_i(\vec{y}_i)$ is another CHC in the system of CHCs, $\mathit{body}_i[\vec{y}_i \mapsto \vec{x}_i]$ is the simultaneous substitution of variables in $\vec{y}_i$ with variables in $\vec{x}_i$ in body_i, and $\mathit{fresh}(e, \vec{x}_i, C)$ is the result of replacing each variable in e that does not occur in $\vec{x}_i$ with a variable not in C. We call the result of unfolding an uninterpreted predicate in a CHC C (possibly many times) an *unfolding* of C.

There may be several CHCs of the form $\mathit{body}_i \Rightarrow R_i(\vec{y}_i)$ within the system of CHCs, leading to several possible unfoldings of uninterpreted predicate R_i in the CHC C.

If U is the body of an unfolding of a query CHC and an under-approximate mapping M is such that $M(U)$ is a satisfiable SMT formula, then the original set of CHCs has no solution, and, furthermore, the satisfying assignment corresponds to an error trace of the original program.

In addition to using unfoldings to generate formulas for demonstrating that a set of CHCs has no solution, unfoldings can be used to help generate subproblems during CHC-based modular program verification. When viewed in terms of program verification, unfolding a predicate roughly corresponds to performing the inlining of the procedure of the original program that the predicate encodes in the system of CHCs.

2.5 CHC Solving

In order to find solutions to systems of CHCs, CHC solvers [84, 112, 103, 39, 95, 149, 74] query to backend SMT (Satisfiability Modulo Theory) solvers [24] to find interpretations that make all CHC rules valid. In addition to classic fixpoint computations, (as in interprocedural dataflow analysis and abstract interpretation) CHC solvers use model checking techniques, e.g., counterexample guided abstraction refinement (CEGAR) [47], interpolation [109], property-directed reachability (PDR) [36, 65], and guess-and-check procedures [72]. They can thus find procedure summaries that are adequate for verification but that are not necessarily least or greatest fixpoints. CHC-based verifiers have been successfully applied to a range of benchmark programs, but there remain significant challenges in handling mutual recursion and in scalability, especially in deductive approaches that rely on unfolding to solve systems of CHCs. Chapter 4 addresses some of these challenges and provides a more comprehensive description of specific CHC solving techniques in Section 4.9.

Many deductive approaches to CHC solving, including the one described in Chapter 4, make use of *unfolding* CHCs [112, 103]. Deductive CHC solvers (and non-CHC-based verifiers) may also rely on *interpolating SMT solvers*, which are able to compute interpolants.

Definition 2.5.1 (Interpolant)**.** Given two formulas $A(V_1 \cup V_2)$ and $B(V_1 \cup V_3)$ where $V_2 \cap V_3 = \varnothing$ and $A(V_1 \cup V_2) \wedge B(V_1 \cup V_3)$ is unsatisfiable, an interpolant for these formulas is a formula $\mathbb{I}(V_1)$ that contains only free variables V_1 shared by A and B, where $A(V_1 \cup V_2) \Rightarrow \mathbb{I}(V_1)$ holds and $\mathbb{I}(V_1) \wedge B(V_1 \cup V_3)$ is unsatisfiable.

```
if (y > 20) {
    while (i < 10) {
        x *= i;
        i++;
    }
} else {
    while (i < 10) {
        x++;
        i++;
    }
}
```

Figure 2.9: Example program

An interpolant for formulas $A(V_1 \cup V_2)$ and $B(V_1 \cup V_3)$ can be viewed as "separating" A and B, and, in verification, can be used to rule out spurious counterexamples when A represents known facts about the program and B represents one or more error traces. In CHC solving, the learned interpolant can often be used to help construct the interpretations in over-approximate mapping O.

2.6 Self-Composition and Product Programs

So far, we have considered only verifying safety properties of programs. The work presented here also considers the verification of certain relational properties of programs, including k-safety hyperproperties, in which error states encompass the states of k copies of a program. Note that a 1-safety property is the same as a non-relational safety property. A *relational verification problem* (RVP) is a tuple consisting of programs $\{P_j\}$, a relational precondition *pre*, and a relational postcondition *post*.

As an ongoing example, let us consider proving hyperproperties about the C-like integer program shown in Figure 2.9. For proving a k-hyperproperty, we use k copies of the program $\{P_j\}$ that are identical modulo renaming and use indices $j \in \{1, \ldots, k\}$ as subscripts to denote variables in the different copies. We assume that each variable initially takes a nondeterministic value in each program copy.

For verifying relational properties of programs, *relational* invariants are useful. For example, let us consider an RVP with precondition $\mathtt{x}_1 < \mathtt{x}_2 \wedge \mathtt{x}_1 > 0 \wedge \mathtt{i}_1 > 0 \wedge \mathtt{i}_1 = \mathtt{i}_2$ (*pre*) and postcondition $\mathtt{x}_1 < \mathtt{x}_2 \vee \mathtt{y}_1 \neq \mathtt{y}_2$ (*post*). For proving this property, knowing that the first loops of P_1 and P_2 are such that, if they execute in lockstep, they maintain the relational invariant that $\mathtt{x}_1 < \mathtt{x}_2$, which would be useful for proving the postcondition.

Rather than developing new formalisms to handle multiple programs during verification, one approach for relational property verification is to compose the different programs or copies of the program or to take their composition, reducing the relational verification problem to a non-relational verification problem over the result of the composition or product [28, 25]. In the resulting program, there will be a set of variables for each of the programs (or program copies). By maintaining a bijective mapping of the composition or product program's variables to the variables of the program (copy) to which they correspond, invariants of the composition or product program can easily be converted to relational invariants and vice-versa. As a result, in later sections, invariants of composition or product programs that correspond to relational invariants over the different programs (or program copies) will also be referred to as relational invariants.

In performing composition, the programs may be composed via sequential composition [148, 28] or parallel composition [143, 152, 21, 31] operators. Using a simplistic sequential composition can prevent the inference of useful relational properties. For example, if composition is performed such that each program copy is run, one after the other, there is no point during execution of the composed program at which intermediate states of one program copy can be related to the intermediate states of another: for the i^{th} program, the first $i-1$ program will have finished their execution and all the $i+1^{th}$ through k^{th} will have yet to start their execution. For example, consider the sequential composition of P_1 and P_2 shown in Figure 2.10. The prop-

```
assume(x_1 < x_2 ∧ i_1 > 0 ∧ i_1 = i_2);
if (y_1 > 20) {
     while (i_1 < 10) {
          x_1 *= i_1;
          i_1++;
     }
} else {
     while (i_1 < 10) {
          x_1++;
          i_1++;
     }
}
if (y_2 > 20) {
     while (i_2 < 10) {
          x_2 *= i_2;
          i_2++;
     }
} else {
     while (i_2 < 10) {
          x_2++;
          i_2++;
     }
}
assert(x_1 < x_2 ∨ y_1 ≠ y_2);
```

Figure 2.10: Sequential composition of P_1 and P_2 from Figure 2.9

erty to check is now a safety property of the composed program, indicated by the assumption of the precondition and assertion of the postcondition. Note that here, in this sequential composition, there is no way to learn about or take advantage of the relational invariant mentioned earlier about the loop bodies of P_1 and P_2; to prove the assertion at the end, individual loop invariants capturing the precise values of i at each iteration will need to be learned separately in order to prove their equality at their end of the composed program. Note that the required invariants will be nonlinear and of the form $x_j = \frac{x_{j,init} \times i_j!}{i_{j,init}!}$ for $j \in \{1, 2\}$, which requires the use of auxiliary variables $x_{j,init}$ and $i_{j,init}$ to denote the initial values of x_j and i_j respectively. Given the difficulty of inferring such invariants, it is preferable to use simpler relational invariants.

With parallel-style composition as used in the Cartesian Hoare Logic approach [143] and other efforts using relational program logics [152, 21, 31], it is possible to try to align program fragments from different copies in order to find useful relational invariants. This alignment, which we call *synchrony*, is also possible to perform during the construction of product programs [25]; in fact, the Cartesian Hoare Logic approach to verification essentially constructs an *implicit* product program during the verification algorithm. Note that with a parallel-style composition, since each program copy is independent, we need only explore a single interleaving, since all interleavings have the same behavior. More elaborate applications of sequential composition that perform some amount of synchrony can avoid the issue presented by a simplistic composition [148], with all sequential compositions being particular interleavings of the result of a parallel composition. Typically, when performing synchrony, the aim is to align program fragments across which useful relational invariants are easy to derive.

A useful interleaving of the parallel composition is shown in Figure 2.11, where here the loops have been aligned so that it is possible here to make use of the fact that corresponding while loops execute in lockstep. The parallel composition has a branch per each possible pair of control-flow decisions. The alignment strategy usedsplits loops so that, for each control-flow decision, loop iterations are executed in lockstep across copies whenever possible using *lockstep composition*, using a *naive composition* that does not perform any synchronization in order to handle the possibility that one loop has finished iterating before the other. In the case of the example program's copies, because all loops iterate the same number of times, the loops resulting from the naive composition actually never iterate, but this is not always the case for programs in general.

Whether naive or lockstep composition is used, it is still necessary to find invariants for the resulting loops and check that they truly are loop invariants. For relational program verification of two programs or program copies, these checks can be formalized as follows, where each loop is encoded as a triple of first-order logic formulas $\langle \mathit{Init}(\vec{x},\vec{u}), \mathit{Guard}(\vec{u}), \mathit{Tr}(\vec{u},\vec{y})\rangle$, where $\mathit{Init}(\vec{x},\vec{u})$ denotes a symbolic encoding of a precondition for the loop, $\mathit{Guard}(\vec{u})$ denotes the encoding of the loop guard, and $\mathit{Tr}(\vec{u},\vec{y})$ encodes the loop body. Here, $\vec{u}$ is the vector of local variables that are live at the loop guard. The *Enc* procedure from Figure 2.5 can be applied directly to the guard and body of the loop to yield $\mathit{Guard}(\vec{u})$ and $\mathit{Tr}(\vec{u},\vec{y})$.

Definition 2.6.1 (Naive composition)**.** Given two loops encoded as triples of first-order logic formulas $\langle \mathit{Init}_1(\vec{x}_1,\vec{u}_1), \mathit{Guard}_1(\vec{u}_1), \mathit{Tr}_1(\vec{u}_1,\vec{y}_1)\rangle$ and $\langle \mathit{Init}_2(\vec{x}_2,\vec{u}_2), \mathit{Guard}_2(\vec{u}_2), \mathit{Tr}_2(\vec{u}_2,\vec{y}_2)\rangle$, a relational precondition $\mathit{pre}(\vec{x}_1,\vec{x}_2)$, and a relational postcondition $\mathit{post}(\vec{y}_1,\vec{y}_2)$, the task of proving that the postcondition holds given the precondition is reduced to the task of finding (individual) inductive invariants $\boldsymbol{I_1}$ and $\boldsymbol{I_2}$:

$$\mathit{pre}(\vec{x}_1,\vec{x}_2) \wedge \mathit{Init}_1(\vec{x}_1,\vec{u}_1) \Rightarrow \boldsymbol{I_1}(\vec{u}_1)$$

$$\mathit{pre}(\vec{x}_1,\vec{x}_2) \wedge \mathit{Init}_2(\vec{x}_2,\vec{u}_2) \Rightarrow \boldsymbol{I_2}(\vec{u}_2)$$

$$\boldsymbol{I_1}(\vec{u}_1) \wedge \mathit{Guard}_1(\vec{u}_1) \wedge \mathit{Tr}_1(\vec{u}_1,\vec{y}_1) \Rightarrow \boldsymbol{I_1}(\vec{y}_1)$$

$$\boldsymbol{I_2}(\vec{u}_1) \wedge \mathit{Guard}_2(\vec{u}_2) \wedge \mathit{Tr}_2(\vec{u}_2,\vec{y}_2) \Rightarrow \boldsymbol{I_2}(\vec{y}_2)$$

$$\boldsymbol{I_1}(\vec{y}_1) \wedge \boldsymbol{I_2}(\vec{y}_2) \wedge \neg \mathit{Guard}_1(\vec{y}_1) \wedge \neg \mathit{Guard}_2(\vec{y}_2) \Rightarrow \mathit{post}(\vec{y}_1,\vec{y}_2)$$

Note that the method of naive composition requires handling of multiple invariants, which is known to be difficult. Furthermore, it might lose some important relational information specified in $\mathit{pre}(\vec{x}_1,\vec{x}_2)$. It is thus preferable to use lockstep composition where possible.

Definition 2.6.2 (Lockstep composition)**.** Given two loops encoded as triples $\langle \mathit{Init}_1(\vec{x}_1,\vec{u}_1), \mathit{Guard}_1(\vec{u}_1), \mathit{Tr}_1(\vec{u}_1,\vec{y}_1)\rangle$ and $\langle \mathit{Init}_2(\vec{x}_2,\vec{u}_2), \mathit{Guard}_2(\vec{u}_2), \mathit{Tr}_2(\vec{u}_2,\vec{y}_2)\rangle$, a relational precondition $\mathit{pre}(\vec{x}_1,\vec{x}_2)$, and a relational postcondition $\mathit{post}(\vec{y}_1,\vec{y}_2)$, let **both loops iterate exactly the same number of times**. Then the task of proving that the postcondition holds given the precondition is reduced to the task of finding one (relational) inductive invariant $\boldsymbol{I}$:

$$\begin{aligned} \mathit{pre}(\vec{x}_1,\vec{x}_2) \wedge \mathit{Init}_1(\vec{x}_1,\vec{u}_1) \wedge \mathit{Init}_2(\vec{x}_2,\vec{u}_2) &\Rightarrow \boldsymbol{I}(\vec{u}_1,\vec{u}_2) \\ \boldsymbol{I}(\vec{u}_1,\vec{u}_2) \wedge \mathit{Guard}_1(\vec{u}_1) \wedge \mathit{Tr}_1(\vec{u}_1,\vec{y}_1) \wedge \mathit{Guard}_2(\vec{u}_2) \wedge \mathit{Tr}_2(\vec{u}_2,\vec{y}_2) &\Rightarrow \boldsymbol{I}(\vec{y}_1,\vec{y}_2) \\ \boldsymbol{I}(\vec{y}_1,\vec{y}_2) \wedge \neg \mathit{Guard}_1(\vec{y}_1) \wedge \neg \mathit{Guard}_2(\vec{y}_2) &\Rightarrow \mathit{post}(\vec{y}_1,\vec{y}_2) \end{aligned}$$

For the composition shown in Figure 2.11, a Hoare-style analysis can help yield the loop invariant $\mathtt{x}_1 < \mathtt{x}_2 \wedge \mathtt{i}_1 = \mathtt{i}_2$ for the first while loop, which is the loop resulting from lockstep composition. Further forward analysis makes it apparent that the following loops within the first branch will not iterate even once, finally allowing such a Hoare-style analysis to prove the assertion holds for this case, having only needed the single invariant for the lockstep composition.

For the second and third branches, the only loop invariants needed within these branches are the invariants that $\mathtt{y}_1 \neq \mathtt{y}_2$, and for the final branch, again the invariant $\mathtt{x}_1 < \mathtt{x}_2 \wedge \mathtt{i}_1 = \mathtt{i}_2$ holds for the first while loop (which again results from a lockstep composition) with the remaining loops again not iterating even a single time. Using synchrony to explore this particular interleaving of a parallel composition thus allows a Hoare-style analysis to show that the assertion holds for the composed program.

There are challenges in achieving such synchrony and inferring and applying relational invariants in a fully automated setting. In particular, in order to make sure relational invariants are inferred for loops where possible, detecting which loops can be executed in lockstep becomes important. We will describe such challenges in more detail in the following chapter.

In addition to performing composition or constructing a product program directly, completely CHC-based approaches perform synchrony directly on CHCs by using predicate pairing [60, 113, 59]. Predicates in the original system of CHCs are paired and a new predicate is introduced, where the new predicate represents the paired predicates' conjunction with each other and an additional constraint. The

```
assume(x₁ < x₂ ∧ i₁ > 0 ∧ i₁ = i₂);
if (y₁ > 20 ∧ y₂ > 20) {
    while (i₁ < 10 ∧ i₂ < 10) {
        x₁ *= i₁; i₁++;
        x₂ *= i₂; i₂++;
    }
    while (i₁ < 10) {
        x₁ *= i₁; i₁++;
    }
    while (i₂ < 10) {
        x₂ *= i₂; i₂++;
    }
} else if (y₁ > 20 ∧ ¬(y₂ > 20)) {
    while (i₁ < 10 ∧ i₂ < 10) {
        x₁ *= i₁; i₁++;
        x₂++; i₂++;
    }
    while (i₁ < 10) {
        x₁ *= i₁; i₁++;
    }
    while (i₂ < 10) {
        x₂++; i₂++;
    }
} else if (¬(y₁ > 20) ∧ y₂ > 20) {
    while (i₁ < 10 ∧ i₂ < 10) {
        x₁++; i₁++;
        x₂ *= i₂; i₂++;
    }
    while (i₁ < 10) {
        x₁++; i₁++;
    }
    while (i₂ < 10) {
        x₂ *= i₂; i₂++;
    }
} else {
    while (i₁ < 10 ∧ i₂ < 10) {
        x₁++; i₁++;
        x₂++; i₂++;
    }
    while (i₁ < 10) {
        x₁++; i₁++;
    }
    while (i₂ < 10) {
        x₂++; i₂++;
    }
}
assert(x₁ < x₂ ∨ y₁ ≠ y₂);
```

Figure 2.11: Synchronized sequential composition of P_1 and P_2 from Figure 2.9

new predicate represents a relational invariant for the program fragments captured by the paired predicates. In the techniques employed in this dissertation, relational invariants are important and used extensively, but synchrony is controlled during verification rather than up-front as is the case with these CHC-based approaches.

Chapter 3

Verification of relational properties

In this chapter, I will describe a compositional framework that leverages relational specifications to simplify generated verification subtasks on a composed program. This framework is driven by two main strategies: synchrony and symmetry. This chapter, unlike subsequent chapters, does not rely on a CHC encoding of programs, and thus does not treat loops as recursive procedures.

The framework assumes that a parallel composition is used to produce a product program, allowing for more potential in aligning loop bodies during verification, and, similar to closely related efforts [25, 143], chooses to synchronize (i.e., align) programs at conditional blocks as well as loops. As with these related efforts, the framework aims to execute loops in lockstep so that relational invariants can be derived over corresponding iterations over the loop bodies, and does so using a novel technique that analyzes relational specifications to infer, under reasonable assumptions, *maximal sets of loops* that can be executed in lockstep. Synchronizing at conditional blocks in addition to loops enables simplification due to relational specifications and conditional guards that might result in infeasible or redundant verification subtasks. Pruning of such infeasible subtasks has been performed and noted as important in existing work [143], and synchronizing at conditional blocks allows us to prune eagerly. More importantly, aligning different programs at conditional statements sets up our next strategy.

The second strategy is the exploitation of symmetry in relational specifications. Due to control flow divergences or non-lockstep executions of loops, even different copies of the same program may proceed along different code fragments. However, some of the resulting verification subtasks may be indistinguishable from each other due to underlying symmetries among related fragments. This strategy analyzes the relational specifications, expressed as formulas in first-order theories (e.g., linear integer arithmetic) with multi-index variables, to discover symmetries and exploit them to prune away redundant subtasks. Prior works on use of symmetry in model checking [67, 46, 96, 64] are typically based on symmetric states satisfying the same set of indexed atomic propositions, and do not consider symmetries among different indices in specifications. In contrast, the work described here is the first to *extract* such symmetries in relational specifications, and to *use* them for pruning redundant subtasks during relational verification. For extracting these symmetries, I have lifted core ideas from symmetry-discovery and symmetry-breaking in SAT formulas [54] to richer formulas in first-order theories.

The strategies proposed for exploiting synchrony and symmetry via relational specifications are fairly general in that they can be employed in various verification methods. I provide a generic logic-based description of these strategies at a high level (Sect. 3.2) and describe a specific instantiation in a verification algorithm based on forward Hoare-style analysis that computes strongest-postconditions (Sect. 3.3). I have implemented this approach in a tool called SYNONYM built on top of the DESCARTES tool [143]. An experimental evaluation (Sect. 3.4) shows the effectiveness of this approach in improving the performance of verification in many examples (and a marginal overhead in smaller examples). In particular, exploiting symmetry is crucial

$$
\begin{gathered}
y_1 > 20 \wedge y_2 > 20 \wedge y_3 > 20 \\
y_1 > 20 \wedge y_2 > 20 \wedge y_3 \leq 20 \\
y_1 > 20 \wedge y_2 \leq 20 \wedge y_3 > 20 \\
y_1 > 20 \wedge y_2 \leq 20 \wedge y_3 \leq 20 \\
y_1 \leq 20 \wedge y_2 > 20 \wedge y_3 > 20 \\
y_1 \leq 20 \wedge y_2 > 20 \wedge y_3 \leq 20 \\
y_1 \leq 20 \wedge y_2 \leq 20 \wedge y_3 > 20 \\
y_1 \leq 20 \wedge y_2 \leq 20 \wedge y_3 \leq 20
\end{gathered}
$$

Figure 3.1: Eight possible control-flow decisions

in enabling verification to complete for some properties, without which DESCARTES exceeds a timeout on all benchmark examples.

The work presented in this chapter has previously been presented and published at a conference [123].

3.1 Motivating Example

Consider three program copies $\{P_j\}$ of the program from Figure 2.9. We can use these three program copies to help prove a 3-safety hyperproperty about the program. A variety of useful properties can be expressed as k-safety properties with $k \geq 3$, such as transitivity and homomorphism (both 3-safety properties), as well as associativity (a 4-safety property) [143].

Recall that a *relational verification problem* (RVP) is a tuple consisting of programs $\{P_j\}$, a relational precondition *pre*, and a relational postcondition *post*. In the example RVPs below involving 3-safety properties, we will consider the three conditionals, which in turn lead to eight possible control-flow decisions (Figure 3.1) in a composed program. Each RVP reduces to subproblems for proving that *post* can be derived from *pre* for each of these control-flow decisions. In the rest of the section, I demonstrate the underlying ideas behind the proposed approach to solve these subproblems efficiently.

Maximizing Lockstep Execution. Given an RVP (referred to as RVP_1) with precondition $x_1 < x_3 \wedge x_1 > 0 \wedge i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$ (*pre*) and postcondition $(x_1 < x_3 \vee y_1 \neq y_3) \wedge i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$ (*post*), consider a control-flow decision $y_1 > 20 \wedge y_2 > 20 \wedge y_3 > 20$. This leads to another RVP, consisting of three programs of the following form:

```
assume(y_j > 20); while (i_j < 10) { x_j *= i_j; i_j++; }
```

where $j \in \{1, 2, 3\}$, and the aforementioned *pre* and *post*. From *pre*, it follows that $i_1 = i_3$ and $i_2 \geq i_1$. We can thus infer that the first and third loops are always executed the same number of times, while the second loop may be executed for fewer iterations. This knowledge allows a program analyzer to infer a single relational invariant for the first and third loops and handle the second loop separately. Clearly, the relational invariant $x_1 < x_3 \wedge i_1 = i_3 \wedge i_1 \leq 10$ and the non-relational invariant $i_2 \leq 10$ are enough to derive *post*. If we were to handle the first and third loop separately, we would need the complex nonlinear invariants per while loop as described in Sect. 2.6.

Previous approaches could not guarantee that the first and third loops of our example are always analyzed in lockstep. A naive self-composition approach never performs lockstep execution of loops, and must always infer a non-relational invariant for each loop. The approach employed by the DESCARTES [143] algorithm improves upon this naive self-composition approach by attempting lockstep execution; however, when all loops cannot be executed in lockstep, the algorithm selects an arbitrary loop to handle independently before trying again. Here, if the first loop or third loop is selected, then because the remaining loops cannot be executed in lockstep, DESCARTES must, like the naive self-composition approach, infer non-relational invariants for each loop. Sect. 3.2.1 describes my proposed algorithm for maximizing the number of loops to be executed in lockstep that can help avoid inference of potentially difficult non-relational invariants.

Symmetry-Breaking. For the same program, and an RVP (referred to as RVP_2) with precondition $i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$ and postcondition $i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$, consider a control-flow decision $y_1 > 20 \wedge y_2 > 20 \wedge y_3 \leq 20$. We generate another RVP involving the following set of programs:

```
assume(y1 > 20); while (i1 < 10) { x1 *= i1; i1++; }

assume(y2 > 20); while (i2 < 10) { x2 *= i2; i2++; }

assume(y3 ≤ 20); while (i3 < 10) { x3++; i3++; }
```

Similarly, decision $y_1 \leq 20 \wedge y_2 > 20 \wedge y_3 > 20$ generates yet another RVP over the following:

```
assume(y1 ≤ 20); while (i1 < 10) { x1++; i1++; }

assume(y2 > 20); while (i2 < 10) { x2 *= i2; i2++; }

assume(y3 > 20); while (i3 < 10) { x3 *= i3; i3++; }
```

Both RVPs have the same precondition and postcondition as RVP_2. It is evident that both RVPs differ only in their subscripts; by taking one and swapping the subscripts 1 and 3 due to symmetry, we arrive at the other. Thus, we can consider them equivalent RVPs. Knowing the verification result for either RVP allows us to skip verifying the other one by exploiting such symmetries.

3.2 Leveraging Relational Specifications

In this section, I describe the main components of the compositional framework, which leverage relational specifications to simplify the verification subtasks. I first describe a novel algorithm for inferring maximal sets of loops that can be executed in lockstep (Sect. 3.2.1). Next, I describe a technique for handling conditionals (Sect. 3.2.2). While this is similar to other prior work, the main purpose here is to set the stage for our novel methods for exploiting symmetry (Sect. 3.2.3). For each of these compo-

Algorithm 1 Procedure for finding sets of loops to execute in lockstep

```
1:  procedure CheckLockstep(set of programs $\{P_1, \ldots, P_k\}$)
2:      $\mathit{Loops} \leftarrow$ GetLoopPerProgram($P_1, \ldots, P_k$)
3:      $\mathit{Inv} \leftarrow$ InferInvariant($\mathit{Loops}$)
4:      $\mathit{Query} \leftarrow$ ConstructQuery($\mathit{Inv}$, $\mathit{Loops}$)
5:      if $\mathit{Query}$ has satisfying assignment $M$ then
6:          $\mathit{Terminated} \leftarrow \varnothing$
7:          $\mathit{Unfinished} \leftarrow \varnothing$
8:          for $P_i$ with loop with guard $\mathit{Guard}_i(\vec{u_i})$ do
9:              if $M(\mathit{Guard}_i(\vec{u_i}))$ is false then
10:                 $\mathit{Terminated} \leftarrow \mathit{Terminated} \cup \{P_i\}$
11:             else
12:                 $\mathit{Unfinished} \leftarrow \mathit{Unfinished} \cup \{P_i\}$
13:         return CheckLockstep($\mathit{Terminated}$) $\cup$ CheckLockstep($\mathit{Unfinished}$)
14:     return $\{\{P_1, \ldots, P_k\}\}$
```

nents, the framework relies on the relational precondition (and sometimes the postcondition) to help simplify the verification tasks.

3.2.1 Synchronizing loops

Given a set of programs containing loops, the goal is to determine which ones can be executed in lockstep. As demonstrated previously, relational invariants over lockstep loops are often easier to derive than loop invariants over a single copy.

The algorithm CheckLockstep shown in Algorithm 1 takes as input a set of programs with loops $\{P_1, \ldots, P_k\}$ and outputs a set of *maximal* classes of programs with loops that can be executed in lockstep. The algorithm partitions its input set of programs and recursively calls CheckLockstep on the partitions.

First, CheckLockstep gets the loops to be considered for each program and infers a relational inductive invariant over the loop bodies by calling InferInvariant, which synthesizes and returns $\boldsymbol{I}(\vec{u}_1, \ldots, \vec{u}_k)$ in the following for the k loops provided as arguments, each encoded as a triple $\langle \mathit{Init}_i(\vec{x}_i, \vec{u}_i), \mathit{Guard}_i(\vec{u}_i), \mathit{Tr}_i(\vec{u}_i, \vec{y}_i)\rangle$ as described in Chapter 2:

$$\begin{aligned} \mathit{pre}(\vec{x}_1, \dots, \vec{x}_k) \wedge \bigwedge_{i=1}^{k} \mathit{Init}_i(\vec{x}_i, \vec{u}_i) &\implies \boldsymbol{I}(\vec{u}_1, \dots, \vec{u}_k) \\ \boldsymbol{I}(\vec{u}_1, \dots, \vec{u}_k) \wedge \bigwedge_{i=1}^{k} \mathit{Guard}_i(\vec{u}_i) \wedge \mathit{Tr}_i(\vec{u}_i, \vec{y}_i) &\implies \boldsymbol{I}(\vec{y}_1, \dots, \vec{y}_k) \end{aligned}$$

CheckLockstep then poses the following query:

$$\neg\left(\left(\boldsymbol{I}(\vec{u}_1, \dots, \vec{u}_k) \wedge \bigvee_{i=1}^{k} \neg \mathit{Guard}_i(\vec{u}_i)\right) \implies \bigwedge_{i=1}^{k} \neg \mathit{Guard}_i(\vec{u}_i)\right) \tag{3.1}$$

The left-hand side of the implication holds whenever one of the loops has terminated (the relational invariant holds and at least one of the loop conditions must be false), and the right-hand side holds only if all of the loops have terminated. If the formula is unsatisfiable, then the termination of one loop implies the termination of all loops, and all loops can be executed simultaneously [143]. In this case, the entire set of input programs is one maximal class, and the set containing the set of all input programs is returned.

Otherwise, CheckLockstep gets a satisfying assignment and partitions the input programs into a set *Terminated* and a set *Unfinished*. The *Terminated* set contains all programs P_i whose guards $\mathit{Guard}(\vec{u}_i)$ are false in the model for the formula, and the *Unfinished* set contains the remaining programs. The CheckLockstep algorithm is then called recursively on both *Terminated* and *Unfinished*, with its final result being the union of the two sets returned by these recursive calls.

The following theorem assumes that any relational invariant $\boldsymbol{I}(\vec{u}_1, \dots, \vec{u}_k)$, generated externally and used by the algorithm, is stronger than any relational invariant $\boldsymbol{I}(\vec{u}_1, \dots, \vec{u}_{i-1}, \vec{u}_{i+1}, \dots, \vec{u}_k)$ that could be synthesized over the same set of k loops with the i^{th} loop removed.

Theorem 3.2.1. *For any call to* CHECKLOCKSTEP*, it always partitions its set of input programs such that for all $P_i \in \mathit{Terminated}$ and $P_j \in \mathit{Unfinished}$, P_i and P_j cannot be executed in lockstep.*

Proof. Assume that CHECKLOCKSTEP has partitioned its set of programs into the *Terminated* and *Unfinished* sets. Let $P_i \in \mathit{Terminated}, P_j \in \mathit{Unfinished}$ be arbitrary programs. Based on how the partitioning is performed, we know that there is a model for Eq. 3.1 such that $\mathit{Guard}_i(\vec{u}_i)$ does not hold and $\mathit{Guard}_j(\vec{u}_j)$ does. We can thus conclude that the following formula is satisfiable:

$$\neg\Big(\boldsymbol{I}(\vec{u}_1, \ldots, \vec{u}_k) \wedge \neg \mathit{Guard}_i(\vec{u}_i) \implies \neg \mathit{Guard}_j(\vec{u}_j)\Big)$$

From the assumption on the invariant synthesizer, we conclude that the following is also satisfiable, indicating that P_i and P_j cannot be executed in lockstep:

$$\neg\Big(\boldsymbol{I}(\vec{u}_i, \vec{u}_j) \wedge \neg \mathit{Guard}_i(\vec{u}_i) \implies \neg \mathit{Guard}_j(\vec{u}_j)\Big)$$

where $\boldsymbol{I}(\vec{u}_i, \vec{u}_j)$ is the relational invariant for P_i and P_j that the invariant synthesizer infers. □

3.2.2 Synchronizing conditionals

Let two programs have forms `if` $\mathtt{Q}_i$ `then` $\mathtt{R}_i$ `else` $\mathtt{S}_i$, where $i \in \{1, 2\}$ and $\mathtt{R}_i$ and $\mathtt{S}_i$ are arbitrary blocks of code and could possibly have loops. Let them be a part of some RVP, which generates four verification subproblems, where each RVP generated corresponds to one of the four possible control-flow decisions. As seen in previous sections, each of the four verification tasks could be expensive. In order to reduce the number of verification tasks where possible, the verification approach presented here uses the relational preconditions to filter out pairs of programs for which verification conclusions can be derived trivially.

For k programs of the form `if` $\mathtt{Q}_i$ `then` $\mathtt{R}_i$ `else` $\mathtt{S}_i$ for $i \in \{1, \ldots, k\}$ and precondition $\mathit{pre}(\vec{x}_1, \ldots, \vec{x}_k)$, the verifier can simultaneously generate all possible combinations of decisions by querying a solver for all truth assignments to the $\mathtt{Q}_i$s:

$$pre(\vec{x}_1, \ldots, \vec{x}_k) \wedge \bigwedge_{i=1}^{k} Q_i \tag{3.2}$$

The result of this All-SAT query is used to generate sets of programs in subtasks. For each assignment j, where each Q_i is assigned a Boolean value v_i, the following set is generated: $\{\mathtt{assume(V_1);\ U_1}, \ldots,\ \mathtt{assume\ (V_k);\ U_k}\}$ where for each $i \in \{1, \ldots, k\}$, if $v_i = \top$, then $\mathtt{V}_i = \mathtt{Q}_i$ and $\mathtt{U}_i = \mathtt{R}_i$, else $\mathtt{V_i} = \neg\mathtt{Q_i}$ and $\mathtt{U}_i = \mathtt{S}_i$. The verification algorithm need only be applied on the resulting sets of programs. For example, in the above RVP, if $\mathtt{Q}_1$ is equivalent to $\mathtt{Q}_2$ in all solutions, then the RVP reduces to verification of just two pairs of programs:

$$\mathtt{assume(Q_1);\ R_1} \quad \text{and} \quad \mathtt{assume(Q_2);\ R_2}$$

$$\mathtt{assume(\neg Q_1);\ S_1} \quad \text{and} \quad \mathtt{assume(\neg Q_2);\ S_2}$$

3.2.3 Discovering and exploiting symmetries

Using the All-SAT query from Eq. 3.2 allows us to prune trivial RVPs. However, as we have seen in Sect. 3.1, some of the remaining RVPs could be regarded as equivalent due to symmetry. First, I discuss how to identify symmetries in formulas syntactically, and then I show how to use such symmetries.

3.2.3.1 Identifying symmetries in formulas

Formally, symmetries in formulas are defined as permutations. Note that any permutation π of set $\{1, \ldots, k\}$ can be lifted to be a permutation of set $\{\vec{x}_1, \ldots, \vec{x}_k\}$.

Definition 3.2.1 (Symmetry)**.** Let $\vec{x}_1, \ldots, \vec{x}_k$ be vectors of the same size over disjoint sets of variables. A *symmetry* π of a formula $F(\vec{x}_1, \ldots, \vec{x}_k)$ is a permutation of set $\{\vec{x}_i \mid 1 \leq i \leq k\}$ such that $F(\vec{x}_1, \ldots, \vec{x}_k) \iff F(\pi(\vec{x}_1), \ldots, \pi(\vec{x}_k))$.

The task of finding symmetries within a set of formulas can be performed syntactically by first canonicalizing the formulas, converting the formulas into a graph

Algorithm 2 Algorithm for constructing a graph to find symmetries.

```
1: procedure MakeGraph($F$)
2:     $(V, E) \leftarrow (\{v_1^{Id}, \ldots, v_k^{Id}\}, \varnothing)$ where each $v_i^{Id}$ has $color(v_i^{Id}) = Id$
3:     for $d \in$ Clauses($F$) do $(V, E) \leftarrow$ MakeColoredAST($d$) $\cup (V, E)$
4:     for $v \in V$ with $x_i \in vars(color(v))$ do
5:         $V \leftarrow (V \setminus \{v\}) \cup \{$Recolor$(v, v[x_i \mapsto x])\}$
6:         $E \leftarrow E \cup \{(v, v_i^{Id})\}$
```

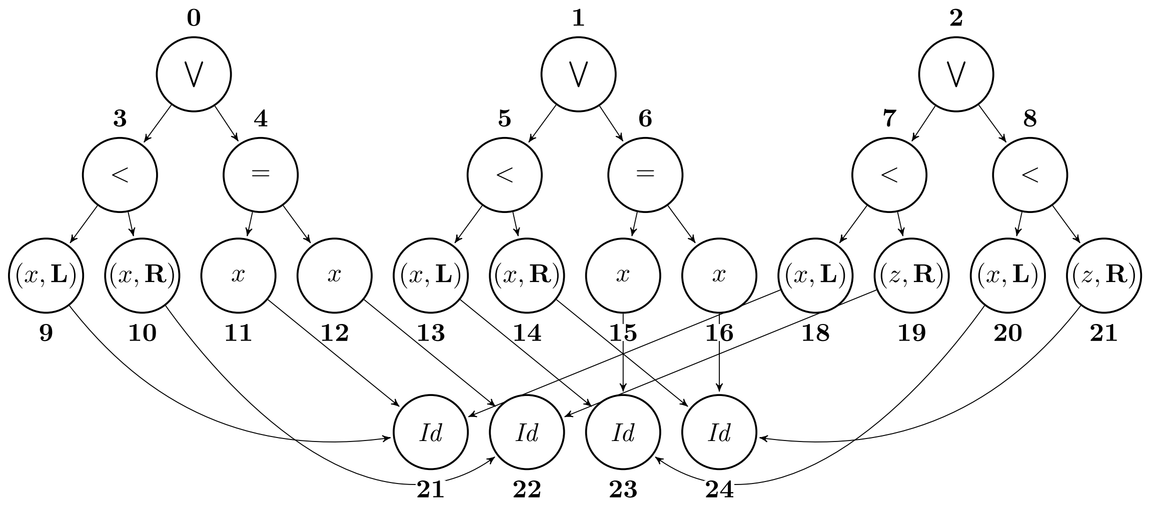


Figure 3.2: Graph with vertex names (outside the vertices) and colors (inside the vertices).

representation of their syntax, and then using a graph automorphism algorithm to find the symmetries of the graph. I demonstrate how this can be done for a formula φ over Linear Integer Arithmetic with the following example.

Let $\varphi = (x_1 \leq x_2 \wedge x_3 \leq x_4) \wedge (x_1 < z_2 \vee x_3 < z_4)$. Note that this formula is symmetric under a permutation of the subscripts that simultaneously swaps 1 with 3 and 2 with 4. Let $\{(x_1, z_1), (x_2, z_2), (x_3, z_3), (x_4, z_4)\}$ be the vectors of variables. A vector is identified by its subscript (e.g., (x_1, z_1) is identified by 1).

The algorithm starts with canonicalizing the formula: $\varphi = (x_1 < x_2 \vee x_1 = x_2) \wedge (x_3 < x_4 \vee x_3 = x_4) \wedge (x_1 < z_2 \vee x_3 < z_4)$. It then constructs a colored graph for the canonicalized formula with the procedure in Algorithm 2. The algorithm initializes a graph by the set of k vertices $v_1^{Id}, \ldots, v_k^{Id}$ with color Id (vertices 21-24 in Figure 3.2), where k is the number of identifiers. It then (Line 3) adds to the graph the union of the abstract syntax trees (AST) for the formula's conjuncts, where each vertex has a color corresponding to the type of its AST node. If a parent vertex has a color of an ordering-sensitive operation or predicate, then the children should have colors that include a tag to indicate their ordering (e.g., vertices 9 and 10 in Figure 3.2 have colors with tags because their parent has color $<$, but vertices 11 and 12 do not have tags because their parent has color $=$). Next (Line 4), the algorithm performs an appropriate renaming of vertex colors so that each indexed variable name x_i is replaced with a non-indexed version x, while simultaneously adding edges from each vertex with a renamed color to v_i^{Id}. The resulting graph for φ is shown in Figure 3.2. Finally, the algorithm applies a graph automorphism finder to get the following automorphism (in addition to the identity automorphism), which is shown here in a cyclic notation where $(x\ y)$ means that $x \mapsto y$ and $y \mapsto x$ (vertices that map to themselves are omitted):

$$(0\ 1)(3\ 5)(4\ 6)(7\ 8)(9\ 13)(10\ 14)(11\ 15)(12\ 16)(17\ 19)(18\ 20)(21\ 23)(22\ 24)$$

Only permutations of the vectors are of interest, so the algorithm projects out the relevant parts of the permutation (21 23)(22 24) and maps them back to the vector identifiers. The result is the following permutation on the identifiers:

$$\pi = \{1 \mapsto 3, 2 \mapsto 4, 3 \mapsto 1, 4 \mapsto 2\}$$

3.2.3.2 Exploiting symmetries

I now define the notion of symmetric RVPs and demonstrate the application of symmetry-breaking to generate a single representative per equivalence class of RVPs.

Definition 3.2.2 (Symmetric RVPs)**.** Two RVPs:

$\langle Ps, pre(\vec{x}_1, \dots, \vec{x}_k), post(\vec{y}_1, \dots, \vec{y}_k)\rangle$ and $\langle Ps', pre(\vec{x}_1, \dots, \vec{x}_k), post(\vec{y}_1, \dots, \vec{y}_k)\rangle$, where $Ps = \{P_1, \dots, P_k\}$, and $Ps' = \{P'_1, \dots, P'_k\}$, are called *symmetric* under a permutation π iff

1. π is a symmetry of formula $\mathit{pre}(\vec{x}_1, \ldots, \vec{x}_k) \wedge \mathit{post}(\vec{y}_1, \ldots, \vec{y}_k)$

2. for every $P_i \in Ps$ and $P_j \in Ps'$, if $\pi(i) = j$, then P_i and P_j have the same number of inputs and outputs and have logically equivalent encodings for the same set of input variables $\vec{x}_i$ and output variables $\vec{y}_i$

As seen in Sect. 3.2.3.1, identification of symmetries could be made purely on the syntactic level of the relational preconditions and postconditions. For each detected symmetry, it remains to check equivalence between the corresponding programs' encodings, which can be formulated as an SMT problem.

I will describe a simple but intuitive approach that allows the approach to exploit symmetries. First, identify the set of symmetries using $\mathit{pre} \wedge \mathit{post}$. Then, solve the All-SAT query from Eq. 3.2 and get a *reduced* set R of RVPs (i.e., one without all trivial problems). For each $RVP_i \in R$, perform the relational verification only if no symmetric $RVP_j \in R$ has already been verified. Thus, the most expensive part of the routine, checking equivalence of RVPs, is performed on demand and only on a subset of all possible pairs $\langle RVP_i, RVP_j \rangle$.

Alternatively, in some cases (e.g., for parallelizing the algorithm) it might help to identify all symmetric RVPs prior to solving the All-SAT query from Eq. 3.2. From this set, it is possible to generate symmetry-breaking predicates (SBPs) [54] and conjoin them to Eq. 3.2. Constrained with SBPs, this query will have fewer models, and will contain a single representative per equivalence class of RVPs. I describe how to construct SBPs in more detail in the next section.

3.2.3.3 Generating Symmetry-Breaking Predicates (SBPs)

SBPs have previously been applied in pruning the search space explored by SAT solvers [54, 7]. Traditionally, techniques construct SBPs based on symmetries in truth assignments to the literals in the formula, but SBP-construction can be adapted to be based on symmetries in truth assignments to conditionals, allowing for symmetry-breaking in the setting considered here.

An SBP can be constructed by treating each condition the way a literal is treated in existing SBP constructions. In particular, it is possible to construct the common Lex-Leader SBP used for predicate logic [54], which in this setting will force a solver to choose the lexicographically least representative per equivalence class for a particular ordering of the conditions. For the ordering of conditions where $\mathtt{Q}_i \leq \mathtt{Q}_j$ iff $i \leq j$ and a set of symmetries S over $\{1, \ldots, k\}$, it is possible to construct a Lex-Leader SBP $SBP(S) = \bigwedge_{\pi \in S} PP(\pi)$ with the more efficient predicate chaining construction [7], where it is the case that

$$PP(\pi) = p_{\min(I)} \wedge \bigwedge_{i \in I} p_i \implies g_{prev(i,I)} \implies l_i \wedge p_{next(i,I)}$$

and that I is the support of π with the last condition for each cycle removed, $\min(I)$ is the minimal element of I, $prev(i, I)$ is the maximal element of I still less than i or 0 if there is none, $next(i, I)$ is the minimal element of I still greater than i or 0 if there is none, $p_0 = g_0 = \top$, p_i is a fresh predicate for $i \neq 0$, $g_i = \mathtt{Q}_{\pi(i)} \implies \mathtt{Q}_i$ for $i \neq 0$, and $l_i = \mathtt{Q}_i \implies \mathtt{Q}_{\pi(i)}$.

After the SBP is constructed, it can be conjoined to the All-SAT query in Eq. 3.2. The solver now generates sets of programs that, when combined with the relational precondition and postcondition, form a set of irredundant RVPs.

Example. Let us consider how SBPs can be applied to RVP_2 from Sect. 3.1 to avoid generating two of the eight RVPs we would otherwise generate.

First, we see that our three programs are all copies of the same program and are at the same program point, so they will have the same encoding. Next, we find the set of permutations S over $\{1, 2, 3\}$ such that for each $\pi \in S$, we have that $i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$ iff $i_{\pi(1)} > 0 \wedge i_{\pi(2)} \geq i_{\pi(1)} \wedge i_{\pi(1)} = i_{\pi(3)}$. In this case, we have that S is the set of permutations $\{\{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3\}, \{1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 3\}\}$. Now,

Algorithm 3 Procedure for solving relational verification problems.

```
1: procedure VERIFY($pre, Current, Ifs, Loops, post$)
2:     while $Current \neq \varnothing$ do
3:         if PROCESSSTATEMENT($pre, P_i, Ifs, Loops, post$) = safe then return safe
4:     if $Loops \neq \varnothing$ then HANDLELOOPS($pre, Loops, post$)
5:     else if $Ifs \neq \varnothing$ then HANDLEIFS($pre, Ifs, Loops, post$)
6:     else return unsafe
```

we construct a Lex-Leader SBP (using the predicate chaining construction described above):

$$p_1 \quad \wedge \quad (p_1 \implies ((y_1 > 20) \implies (y_2 > 20)))$$

where p_1 is a fresh predicate. Conjoining this SBP to Eq. 3.2, leads to the RVPs arising from the control-flow decisions $y_1 > 20 \wedge y_2 > 20 \wedge y_3 \leq 20$ and $y_1 > 20 \wedge y_2 \leq 20 \wedge y_3 \leq 20$ no longer being generated.

3.3 Instantiation of Strategies in Forward Analysis

I now describe an instantiation of the proposed strategies in a verification algorithm based on forward analysis using a strongest-postcondition computation. Other instantiations, e.g., on top of a CHC solver based on Property-Directed Reachability (PDR) [113] are also possible.

Given an RVP in the form of a Hoare triple $\{Pre\}\ P_1 || \cdots || P_k\ \{Post\}$, where $||$ denotes parallel composition, the top-level VERIFY procedure takes as input the relational specification $pre = Pre$ and $post = Post$, the set of input programs $Current = \{P_1, \ldots, P_k\}$, and empty sets $Loops$ and Ifs. It uses a strongest-postcondition computation to compute the next Hoare triple at each step until it can conclude the validity of the original Hoare triple.

Synchronization. Throughout verification, the algorithm maintains three disjoint sets of programs: one for programs that are currently being processed ($Current$), one for programs that have been processed up until a loop (*Loops*), and one for programs that have been processed up until a conditional statement (*Ifs*). The algorithm processes statements in each program independently, with PROCESSSTATEMENT choosing an arbitrary interleaving of statements from the programs in *Current*. When the algorithm encounters the end of a program in its call to PROCESSSTATEMENT, it removes this program from the *Current* set. At this point, the algorithm returns safe if the current Hoare triple is proven valid. When a program has reached a point of control-flow divergence and is processed by PROCESSSTATEMENT, it is removed from *Current* and added to the appropriate set (*Loops* or *Ifs*).

Handling Loops. Once all programs are in the *Loops* or *Ifs* sets (i.e. $\mathit{Current} = \varnothing$), the algorithm handles the programs in the *Loops* set if it is nonempty. HANDLELOOPS behaves like CHECKLOCKSTEP but computes postconditions where possible; when a set of loops are able to be executed in lockstep, HANDLELOOPS computes their postconditions before placing the programs into the *Terminated* set. After all loops have been placed in the *Terminated* set and a new precondition pre' has been computed, rather than returning *Terminated*, HANDLELOOPS invokes $\textsc{Verify}(\mathit{pre}', \mathit{Terminated}, \mathit{Ifs}, \varnothing, \mathit{post})$.

Handling Conditionals. When $\mathit{Current} = \mathit{Loops} = \varnothing$, VERIFY handles conditional statements. HANDLEIFS exploits symmetries by using the All-SAT query with Lex-Leader SBPs as described in Sect. 3.2 and calls VERIFY on each generated verification problem.

3.4 Implementation and Evaluation

To evaluate the effectiveness of increased lockstep execution of loops and symmetry-breaking, I implemented the algorithm from Sect. 3.3 on top of the DESCARTES tool for verifying k-safety properties, i.e., RVPs over k identical Java programs. I implemented two variants: SYN uses only synchrony (i.e., no symmetry is used), while SYNONYM uses both. All implementations (including DESCARTES) use the same guess-and-check invariant generator (the same originally used by DESCARTES, but modified to generate more candidate invariants). SYNONYM computes symmetries in preconditions and postconditions only when all program copies are the same. For this set of examples, it sufficed to compute symmetries simply by checking if each possible permutation leads to equivalent formulas. The implementation includes the syntactic symmetry-finding algorithm from Sect. 3.2.3.1, though it is not used for evaluation here due to its high overhead in using an external tool for finding graph automorphisms. I compare the performance of my implementations to DESCARTES.While there are several tools for relational verification (e.g. ROSETTE/UNBOUND [114], VERIMAPREL [59], REVE [75], MOCHI [75], SYMDIFF [104]), most of these do not handle Java programs, and lack support for k-safety verification for k greater than 2. I use two metrics for comparison: the time taken and the number of Hoare triples processed by the verification procedure. All experiments were conducted on a MacBook Pro, with a 2.7GHz Intel Core i5 processor and 8GB RAM.

3.4.1 Stackoverflow Benchmarks

The first set of benchmarks considered are the Stackoverflow benchmarks originally used to evaluate DESCARTES. These implement (correctly or incorrectly) the Java `Comparator` or `Comparable` interface, and check whether or not their *compare* functions satisfy the following properties:

P1: $\forall x, y. sgn(compare(x, y)) = -sgn(compare(y, x))$

P2: $\forall x, y, z. (compare(x, y) > 0 \land compare(y, z) > 0) \implies compare(x, z) > 0$

P3 $\forall x, y, z. (compare(x, y) = 0) \implies (sgn(compare(x, z)) = sgn(compare(y, z)))$

Table 3.1: Total times (in seconds) and Hoare triple counts (HTC) for Stackoverflow benchmarks. *Improv* reports the total factor of improvement over DESCARTES, where the number of examples is given in parentheses.

Prop	DESCARTES		SYN				SYNONYM			
	Time	HTC	Time	Improv	HTC	Improv	Time	Improv	HTC	Improv
P1	3.11	4422	1.91	1.39 (27)	2255	1.69 (27)	1.82	1.32 (25)	2401	1.82 (32)
			0.57	0.789 (6)	752	0.809 (6)	0.87	0.816 (8)	48	0.979 (1)
P2	24.6	13434	7.83	2.62 (20)	3285	3.081 (16)	7.31	2.80 (19)	3224	3.140 (16)
			4.98	0.823 (13)	4638	0.714 (17)	5.1	0.816 (14)	4638	0.714 (17)
P3	18.85	10938	5.22	2.92 (20)	1565	4.36 (16)	5.22	2.91 (19)	1537	4.74 (16)
			6.18	0.584 (13)	6600	0.623 (17)	6.16	0.594 (14)	6600	0.623 (17)

(One of the original 34 Stackoverflow examples is excluded from the evaluation here because of the inability of the invariant generator to produce a suitable invariant.) I compare the results of running SYN and SYNONYM with the results of running DESCARTES for each property in Figure 3.3 and Figure 3.4 respectively. Examples for which my tools perform as well as or better than DESCARTES are shown in blue and the rest in red. The total times and Hoare triple counts for running DESCARTES and our implementations can be seen in Table 3.1. For each property in the table, the results for SYN and SYNONYM are divided into those for examples where they exhibit a factor of improvement over DESCARTES that is greater or equal to 1 (top) and those for which they do not (bottom).

Because property P1 contains a symmetry, there is a noticeable improvement in terms of number of Hoare triples with the use of symmetry for this property; however, the overhead of computing symmetries leads to SYNONYM performing more slowly than SYN even for some examples that exhibit reduced Hoare triple counts. Property P1 is also the easiest to prove (all implementations can verify each example in under 0.3 seconds), so the overheads contribute more significantly to the runtime. For examples on which our implementations do not perform as well as DESCARTES, we perform reasonably closely to DESCARTES. These examples are typically smaller, and again overheads play a larger role in our poorer performance.

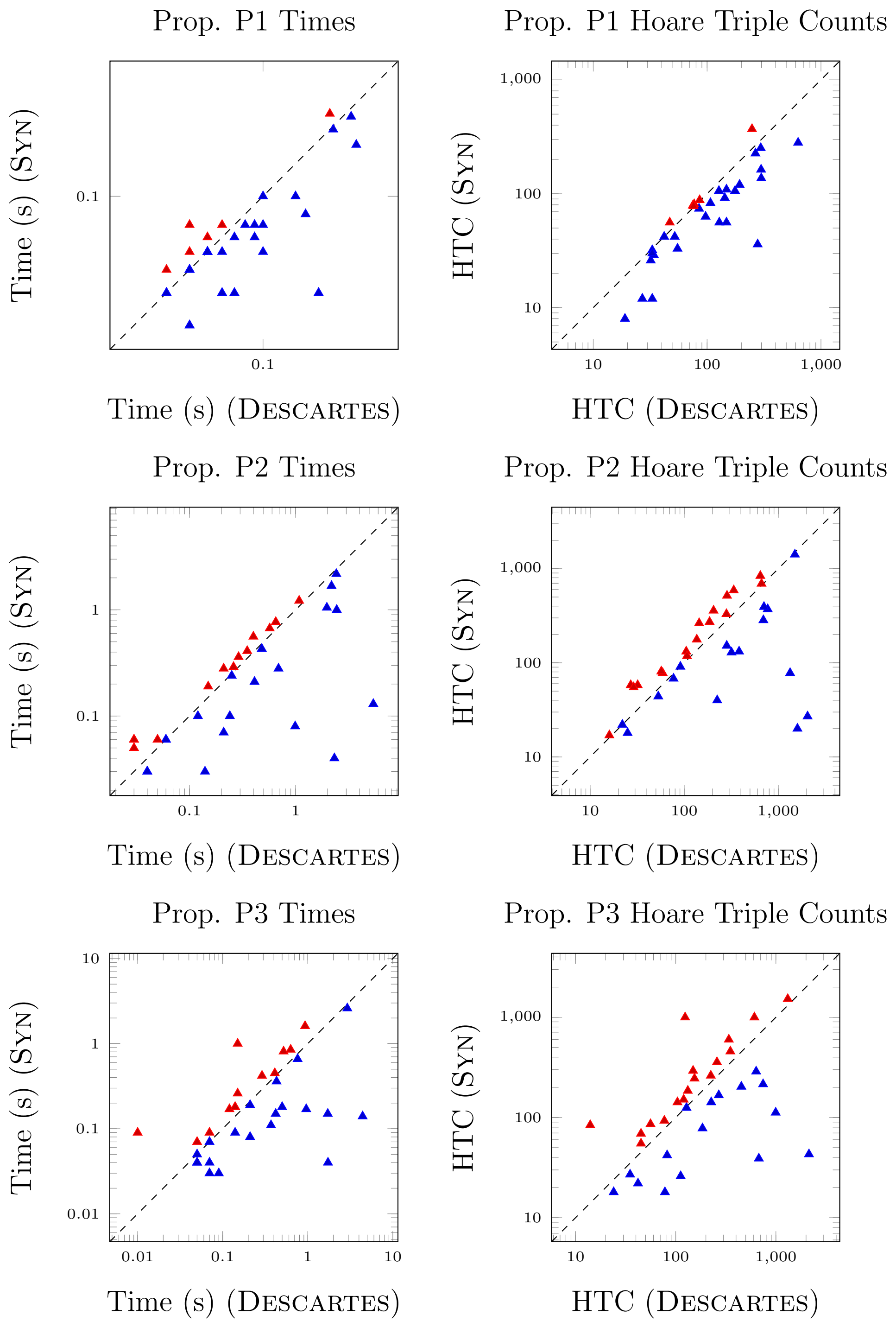


Figure 3.3: Comparison of times and Hoare triple counts between SYN and DESCARTES on Stackoverflow examples.

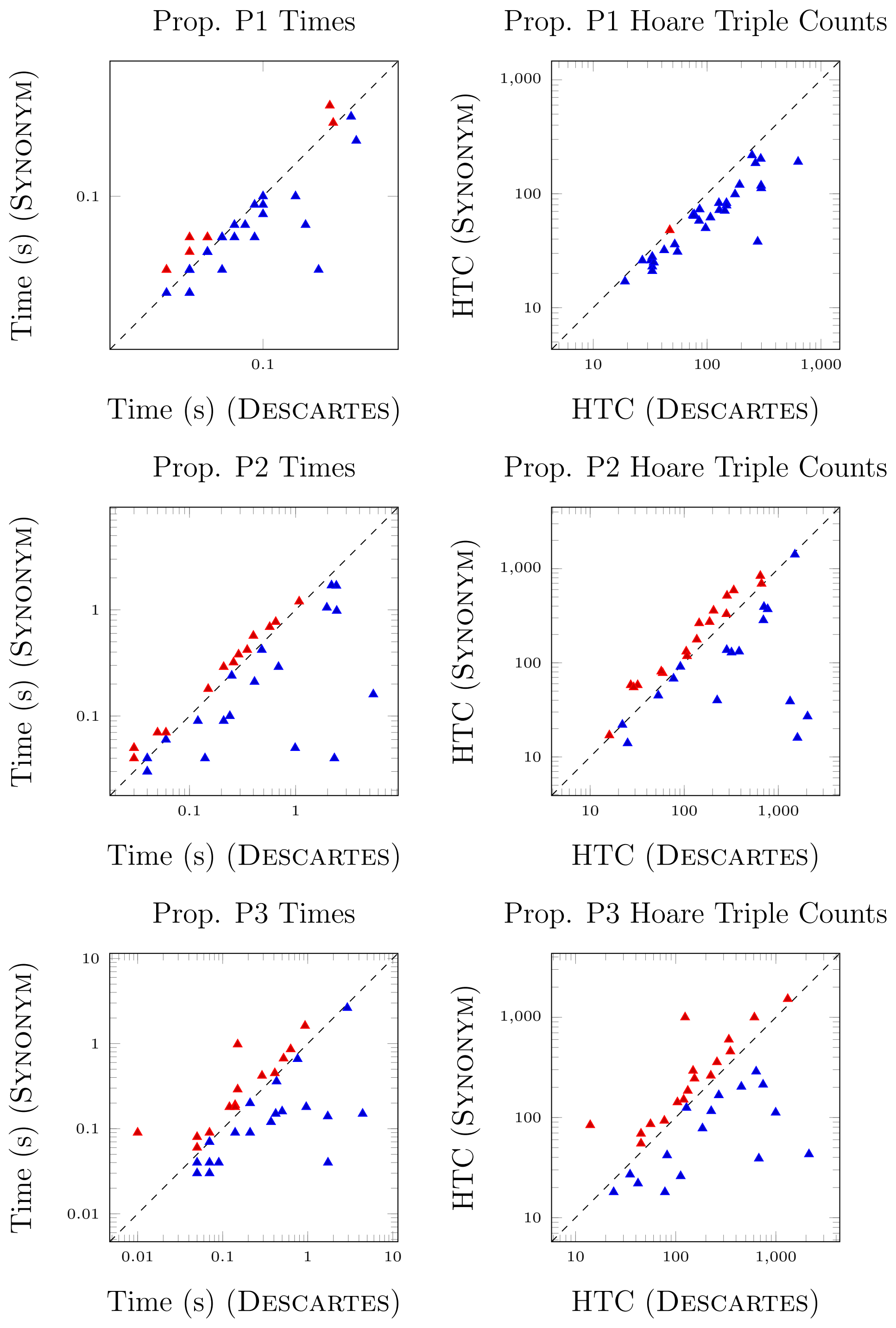


Figure 3.4: Comparison of times and Hoare triple counts between Synonym and Descartes on Stackoverflow examples.

3.4.2 Modified Stackoverflow Benchmarks

The original Stackoverflow examples are fairly small, with all implementations taking under 6 seconds to verify any example. To assess how the tools perform on larger examples, I modified several of the larger Stackoverflow comparator examples to be longer, take more arguments, and contain more control-flow decisions. The resulting functions take three arguments and pick the "largest" object's id, where comparison among objects is performed based on the original Stackoverflow example code. (Ties are broken by choosing the least id.) We check whether these *pick* functions satisfy the following properties that allow reordering input arguments:

P13: $\forall x, y, z. pick(x, y, z) = pick(y, x, z)$

P14: $\forall x, y, z. pick(x, y, z) = pick(y, x, z) \wedge pick(x, y, z) = pick(z, y, x)$

Note that P13 allows swapping the first two input arguments, while P14 allows any permutation of inputs, a useful hyperproperty.

Figure 3.5 shows a comparison of the performance of Syn and Synonym with the results of running Descartes for property P13. Again, examples for which my tools perform as well as or better than Descartes are shown in blue and the rest in red. The total times and Hoare triple counts for running the tools on property P13 are shown in Table 3.2. For these larger examples, Hoare triple counts are more reliably correlated with the time taken to perform verification. Syn outperforms Descartes on 14 of the 16 examples, and Synonym outperforms both Descartes and Syn on all 16 examples.

Figure 3.6 shows a comparison of the performance of Syn and Synonym with the results of running Descartes for property P14. The times and Hoare triple counts for property P14 are shown in Table 3.3. For this property, note that Descartes is unable to verify any of the examples within a one-hour timeout. Meanwhile, Syn is able to verify 10 of the 16 examples without exceeding the timeout. Exploiting

Table 3.2: Verifying P13 for Modified Stackoverflow examples. Times (in seconds) and Hoare triple counts (HTC).

Example	Descartes		Syn		Synonym	
	Time	HTC	Time	HTC	Time	HTC
ArrayInt-pick3-false-simple	1.71	2573	1	1355	0.64	682
ArrayInt-pick3-false	1.55	2591	1.06	1439	0.8	724
ArrayInt-pick3-true-simple	1.71	2573	1.03	1355	0.65	682
ArrayInt-pick3-true	1.55	2591	1.08	1439	0.81	724
Chromosome-pick3-false-simple	0.9	1115	0.9	883	0.53	446
Chromosome-pick3-false	2.51	2891	2.94	3019	1.59	1514
Chromosome-pick3-true-simple	0.9	1115	0.9	883	0.53	446
Chromosome-pick3-true	2.51	2891	2.96	3019	1.59	1514
PokerHand-pick3-false-part1	5.87	5825	0.42	359	0.46	359
PokerHand-pick3-false-part2	9.74	10589	0.85	323	0.86	323
PokerHand-pick3-false	16.91	16475	0.73	159	0.79	159
PokerHand-pick3-true-part1	5.83	5825	3.98	3503	2.4	1756
PokerHand-pick3-true-part2	9.8	10565	7.36	5933	4.53	2971
PokerHand-pick3-true	17.25	16475	12.1	9293	7.34	4651
Solution-pick3-false	76.4	99910	25.05	20645	20.42	10327
Solution-pick3-true	64.5	99910	19.66	20645	15.21	10327
Total	219.64	283914	82.02	74252	59.15	37605
Improvement	1	1	2.68	3.8237	3.713	7.5499

Table 3.3: Verifying P14 for Modified Stackoverflow examples. Times (in seconds) and Hoare triple counts (HTC). - indicates that no sufficient invariant could be inferred.

Example	Descartes		Syn		Synonym	
	Time	HTC	Time	HTC	Time	HTC
ArrayInt-pick3-false-simple	TO	TO	4.12	1938	4.66	1734
ArrayInt-pick3-false	TO	TO	4.92	2017	6.03	1500
ArrayInt-pick3-true-simple	TO	TO	321.15	140593	170.43	58586
ArrayInt-pick3-true	TO	TO	366.98	149125	240.25	62141
Chromosome-pick3-false-simple	TO	TO	47.8	14097	1.67	834
Chromosome-pick3-false	TO	TO	264.21	93052	4.91	3043
Chromosome-pick3-true-simple	TO	TO	299.51	79613	135.56	33179
Chromosome-pick3-true	TO	TO	TO	TO	848.22	225044
PokerHand-pick3-false-part1	TO	TO	0.57	391	0.73	391
PokerHand-pick3-false-part2	TO	TO	0.81	228	0.81	228
PokerHand-pick3-false	-	-	-	-	-	-
PokerHand-pick3-true-part1	TO	TO	2277.03	819553	1272.58	341486
PokerHand-pick3-true-part2	TO	TO	-	-	-	-
PokerHand-pick3-true	-	-	-	-	-	-
Solution-pick3-false	TO	TO	TO	TO	TO	TO
Solution-pick3-true	TO	TO	TO	TO	TO	TO

symmetries here exhibits an obvious improvement, with Synonym not only being able to verify the same examples as Syn, with consistently faster performance on the larger examples, but also being able to verify an additional example within an hour.

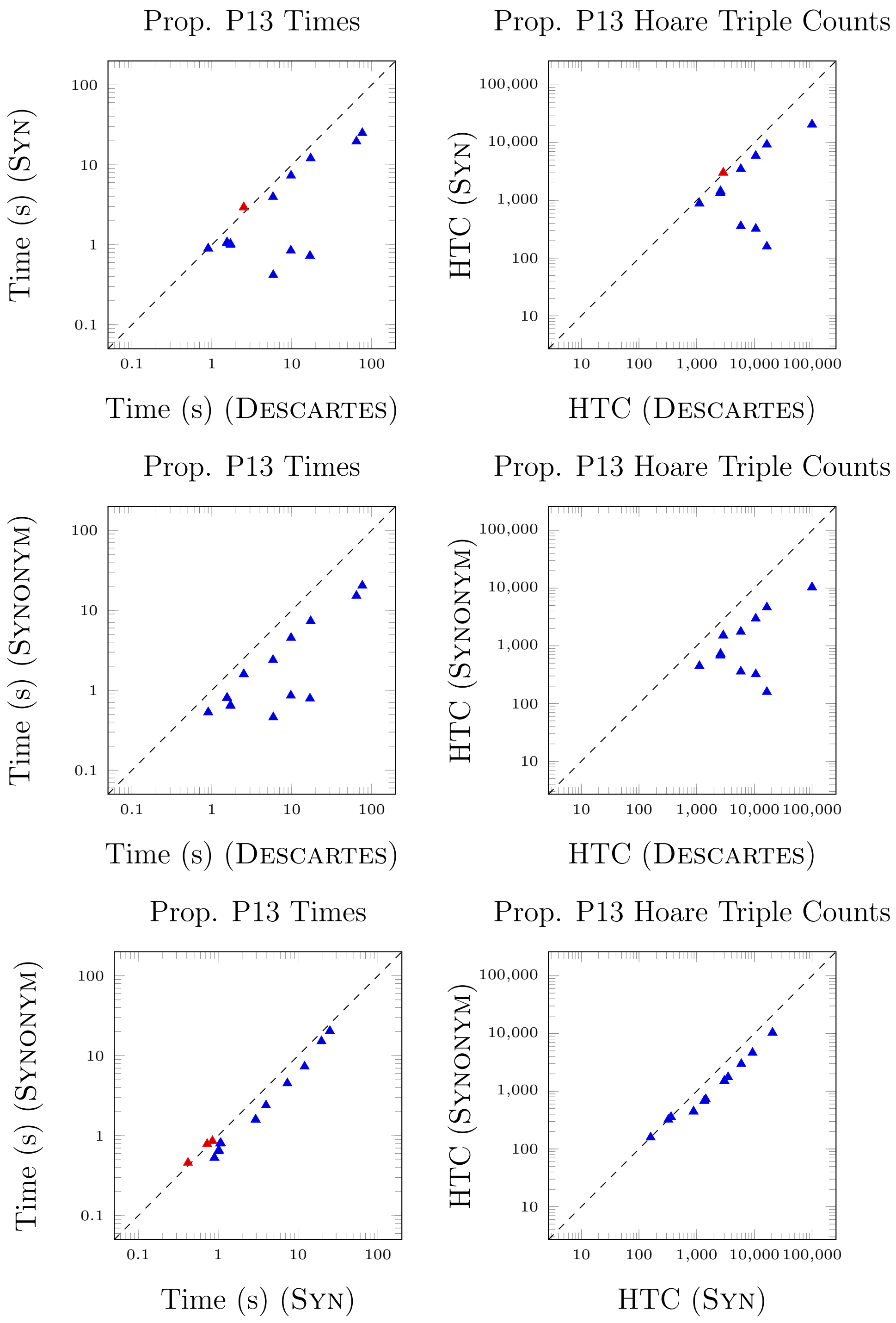


Figure 3.5: Comparison of times and Hoare triple counts between Syn and Descartes, Synonym and Descartes, and Synonym and Syn on Property P13 for modified Stackoverflow examples.

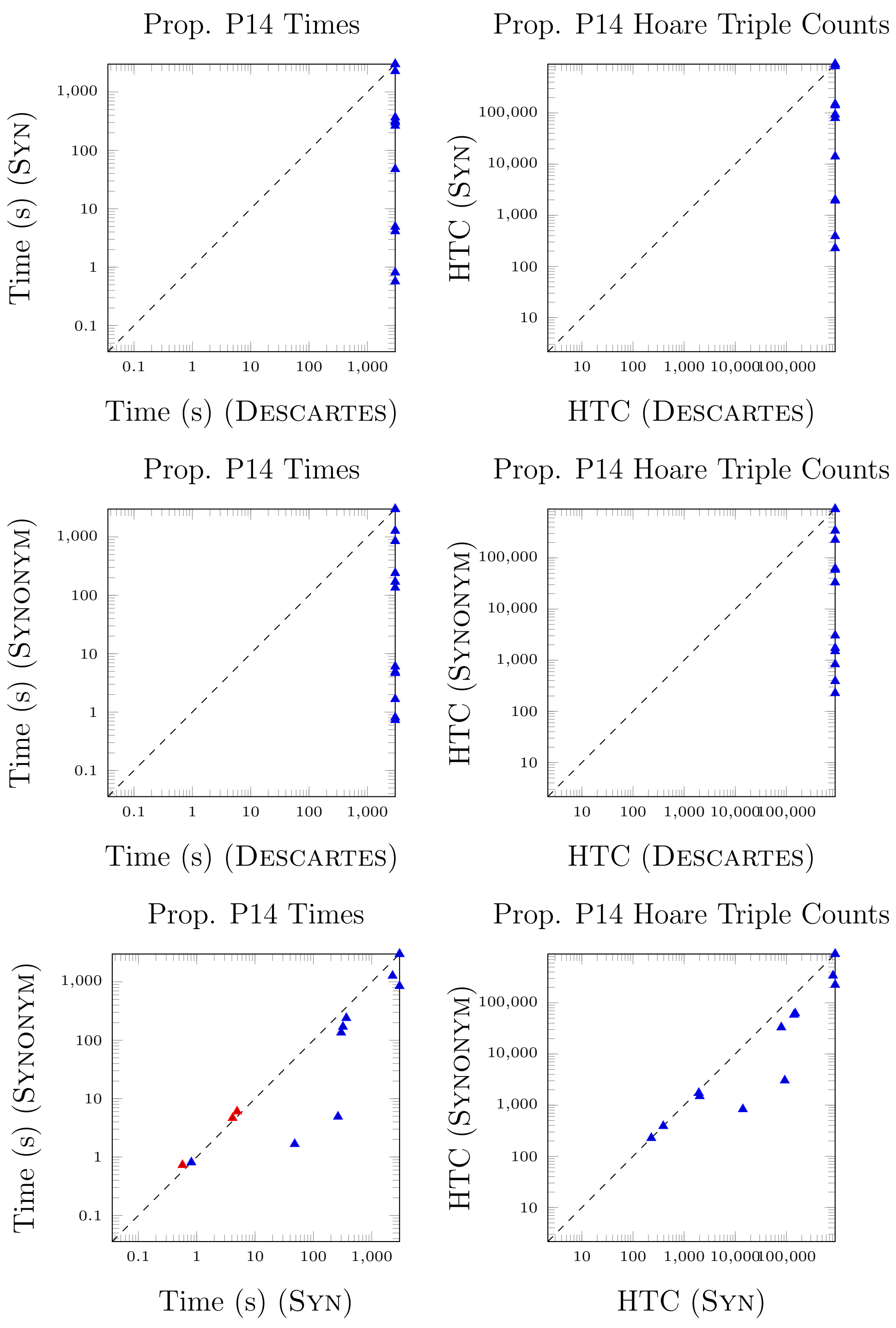


Figure 3.6: Comparison of times and Hoare triple counts between Syn and Descartes, Synonym and Descartes, and Synonym and Syn on Property P14 for modified Stackoverflow examples for which invariants were inferred.

Summary of experimental results. Our experiments indicate that our performance improvements are consistent: on all DESCARTES benchmarks (in Table 3.1, which are all small) our techniques either have low overhead or show some improvement despite the overhead; and on modified (bigger) programs they lead to significant improvements. In particular, we report (Table 3.2) speedups up to 21.4x (on an example where the property doesn't hold) and 4.2x (on an example where it does). More importantly, we report (Table 3.3) that DESCARTES times out on 14 examples, where of these SYNONYM times out for 2 and cannot infer an invariant for one example.

3.5 Related Work

In this section, I will describe work related to what was presented in this chapter. In particular, I will mention related relational and hyperproperty verification techniques, discuss CHL in more detail due to its relevance to the work described in this chapter, and finally describe efforts in using symmetry that inspired the presented symmetry-breaking technique.

3.5.1 Relational and Hyperproperty Verification

As mentioned in Chapter 2, automatic efforts for relational verification reduce relational program verification to safety verification. They typically use composition or program copies or some kind of product program construction [148, 28, 27, 48, 89, 25, 26, 104, 144, 75, 100, 59, 113, 66, 115], with a possible reduction to CHC solving [115, 75, 100, 59, 113]. Similarly to the strategy for synchrony used in SYNONYM, most of them attempt to leverage similarity (structural or functional) in programs to ease verification. Synchronizing parts of programs can be done syntactically [143, 115, 155], which may result in failing to find an alignment for which the underlying solver can find suitable relational invariants. Otherwise, this synchronization can be done semantically [139, 44], in a property-directed or property-aware fashion.

Other relational verification approaches avoid explicitly constructing a product program. Some use program logics to work with Hoare triples that are, in some sense, *relational* [143, 29, 31, 152, 21] or construct product programs implicitly [70]. Still others use decomposition instead of composition [13, 32], employ reinforcement learning [42], or use customized theories with theorem proving [14, 149] instead. While CHL can and has been fully automated, of the many other relational program logics that can be used for reasoning and proving properties about relational programs [31, 152, 3, 21], the majority are used within proof assistants rather than in the context of fully automated verification. Some, such as the work by Banerjee et al. [21], allow for more automation, generating verification conditions that can be translated to and discharged by SMT-based backend tools. Many of these program logics also capture more elaborate program features than, e.g., CHC-based methods can [68, 38], incorporating features from separation logic to reason about the heap [152, 21] or higher-order programs [3], making them harder to automate.

Note that, compared with SYNONYM, these techniques typically have less focus on leveraging relational *specifications* to simplify verification tasks and rather leverage only program structure; even in the case where synchronization is done in a property-directed fashion [139] or where modular relational specifications are taken into account [66, 89], the emphasis is on the semantics of the properties rather than their syntactic structure. Furthermore, none of these techniques consider symmetries among the programs or program copies being considered. The techniques employed by SYNONYM should be able to be applied on top of the mentioned approaches.

There are also some similarities between relational verification and verification of concurrent/parallel programs, as would be expected from the ability to use parallel composition in order to reduce the relational verification problem to a verification problem over the resulting parallel program. In examining these similarities, it is important to note that one does not need to consider different orderings in interleavings of procedures in the composed program for a relational verification problem, provided that the original programs in the relational verification problem are single-threaded. Since the procedures in the composition are independent, it suffices to explore any one ordering. In a concurrent or parallel program, a typical verifier [80, 76] would use *visible* operations (i.e., synchronization operations or communication on shared state) as synchronizing points in the composed program. In the work presented in this chapter as well as other approaches that use synchrony [143, 148], this selection is (ideally) made based on the structure of the component programs and the ease of utilizing or deriving relational assertions for the code fragments.

3.5.2 Cartesian Hoare Logic

The work most closely related to that presented in this chapter is by Sousa and Dillig [143], which proposed Cartesian Hoare Logic (CHL) for proving k-safety properties and the tool DESCARTES for automated reasoning in CHL. In addition to the core program logic, CHL includes additional proof rules for loops, referred to as Cartesian Loop Logic (CLL). A generalization of CHL, called Quantitative Cartesian Hoare Logic was subsequently used by Chen et al. [41] to detect side-channel vulnerabilities in cryptographic implementations.

In terms of comparison, neither CHL nor CLL force alignment at conditional statements or take advantage of symmetries. The algorithm presented for identifying a maximal set of lockstep loops is novel and can be used in other methods that do not rely on CHL/CLL. On the other hand, CLL proof rules allow not only fully lockstep loops, but also *partially* lockstep loops. Although we did not consider it here, the maximal lockstep-loop detection algorithm can be combined with their partial lockstep execution to further improve the efficiency of verification. For example, applying the Fusion 2 rule from CLL to our example while loops generated from RVP_1 (Sect. 3.1) would result in *three* subproblems and require reasoning twice about the second copy's loop finishing later. When combined with maximal lockstep-loop detection, we could generate just *two* subproblems: one where the first and third loops terminate first, and another where the second loop terminates first.

3.5.3 Symmetry

Despite the lack of relational property verifiers that exploit symmetry in programs or in relational specifications, symmetry has been widely applied to other domains. In particular, symmetry has been used very successfully in model checking parametric finite state systems [67, 46, 96] and concurrent programs [64].

The work described here differs from these efforts in two main respects. First, the parametric systems considered in these efforts have components that interact with each other or share variables. Second, the correctness specifications are also parametric, usually single-index or double-index properties in a propositional (temporal) logic. In contrast, in our RVPs, the individual programs are independent and do not share any common variables. The only interaction between them is via relational specifications. Furthermore, we discover symmetries in these relational specifications over multi-index variables, expressed as formulas in first-order theories (e.g., linear integer arithmetic). We then exploit these symmetries to prune redundant RVPs during verification.

Finally, specific applications may impose additional synchrony requirements pertaining to visibility. For example, one may want to check for information leaks from private inputs to public outputs not only at the end of a program but at other specified intermediate points, or information leakage models for side-channel attacks may check for leaks based on given observer models [6]. Such requirements can be viewed as relational specifications at selected synchronizing points in the composed program.

Again, we can leverage these relational specifications to simplify the resulting verification subproblems.

Chapter 4

Verification of safety properties of interprocedural programs

As mentioned in Chapter 2, automated techniques for modular reasoning about interprocedural recursive programs exploit the inherent modularity in a program by deriving a summary for each procedure. In particular, recall that a popular modern approach is to encode interprocedural program verification problems as Constrained Horn Clauses (CHCs) [84], in which uninterpreted predicates represent placeholders for procedure summaries. A CHC solver then finds interpretations for these predicates such that these interpretations correspond to summaries, enabling generation of procedure summaries. In this chapter, I will describe an automated technique for modular reasoning about interprocedural programs that addresses two main challenges in a CHC-style approach to verifying interprocedural programs: the handling of mutually recursive procedures and the scalability of summary inference.

Mutual recursion is a common feature in functional programming and is used widely in programs that traverse over mutually recursive data structures, such as programs that operate on abstract syntax trees or recursive descent parsers. Despite the prevalence and importance of mutual recursion in such domains, prior to this work, existing CHC-solving approaches did not handle mutual recursion well [103, 112, 95, 149, 39, 63, 134]. Solving this challenge addresses a gap in the capabilities of CHC solvers.

The challenge of scalability, mentioned briefly in Chapter 2, is ubiquitous in any verification effort for programs of large enough size. In the context of deductive modular verification and CHC solving, scalability is often pitted against *relevance*, where considering less of the program leads to the inference of procedure summaries that are *less relevant* to proving the property in the specification. Existing deductive techniques for CHC solving tend to sit at one end or the other of the scalability-relevance trade-off [103, 112]; having a way to adjust the trade-off between scalability of summary inference and the relevance of the inferred summaries, as the proposed technique here provides, enhances the landscape of existing deductive CHC-solving techniques.

My proposed technique addresses both of these two challenges with the high-level strategy of leveraging program structure during solving and learning relevant facts. In typical approaches to program verification based on CHC-solving, the program structure may not be maintained when encoding programs as CHCs, especially in a monolithic CHC encoding in which procedures are inlined. As a result, while the proof search proceeds to explore more of the program in each step of CHC solving, the parts that are explored may not reflect any structure in the program call graph. In such cases, although the CHC-solving approach may be modular with respect to a system of CHCs in that it derives each predicate's interpretation separately, this modularity may not correspond to a modular consideration of the original program's procedures. Furthermore, even if the program structure is maintained during the encoding of a program into CHCs, techniques may still ignore program structure during CHC solving.

In contrast, our method both maintains the structure of the call graph in the CHCs during encoding and uses it to guide proof search during CHC solving.

For further improving scalability beyond enabling more modularity within the CHC encoding, our approach ensures that the *backend SMT queries are always bounded in size*, even when more of the program is explored. Bounding the queries' size helps maintain scalability *and* avoid learning over-specialized facts. Bounding is achieved by leveraging the call graph of the program, i.e., analyzing a procedure in the context of a bounded number of levels in the call graph. Furthermore, the notion of a *bounded environment* enables the technique to refer to bounded call paths in the program and learn special lemmas, called *EC (Environment-Call) Lemmas*, which capture relationships among summaries of different procedures on such paths. These lemmas are beneficial in *handling mutual recursion*.

Note that other non-CHC-based program analysis techniques also trade off scalability and relevance by considering a bounded number of levels in a call graph, e.g., in bounded context-sensitivity or k-sensitive pointer/alias analysis [118], stratified inlining [106], and depth cutoff [97] in program verification. However, other than Spacer [103], which is restricted to $k = 1$ bounded environments, existing CHC solvers do not use bounded environments to limit size of the SMT queries.

The work presented in this chapter has previously been presented and published at a conference [125].

4.1 Motivating Example

In this section, I will introduce a motivating example in the form of the program shown in Figure 4.1a. This motivating example will be used in subsequent sections that introduce notions employed by the algorithm, such as derivation trees, bounded contexts, and bounded environments, to provide insights into these notions.

In Figure 4.1a, the procedures `e` and `o` are defined mutually recursively and return `true` iff their argument is respectively even or odd. Procedure `f` returns the (always-even) result of calling `h` on `g`'s result, where `g` returns an arbitrary odd number and `h` adds one to its input. The safety specification is that $\mathtt{e}(\mathtt{f}() - \mathtt{1})$ never holds.

The CHC encoding of this verification problem is shown in Figure 4.1b. We aim to infer interpretations of the predicates in the system of CHCs that correspond to over-approximate procedure summaries. These interpretations should constitute a solution for the system of CHCs. There is a one-to-one mapping from each predicate in the CHC encoding to a procedure of the original program, so I will refer to predicates and procedures interchangeably in the sequel.

Throughout verification, we maintain context-insensitive *over-* and *under-approximate* summaries for all procedures, each of which captures both pre- *and* post-conditions of its procedure. In particular, for each procedure in the system of CHCs, we maintain two mappings: over-approximate mapping O and under-approximate mapping U, where O maps a predicate to an interpretation constituting an over-approximate summary and U maps a predicate to an interpretation constituting an under-approximate summary. All over- (resp. under-) approximate summaries are initially $\top$ (resp. $\bot$), allowing for all (resp. no) behaviors. At each step, we choose a target predicate p and its bounded environment, and then update p's summaries based on the results of SMT queries on over- or under-approximations of bodies of CHCs whose heads contain p. We also allow the bounded environment to be over- or under-approximated, leading to four kinds of SMT queries. These queries let us both over- and under-approximate any predicate, regardless of whether it corresponds to a procedure that is called before or after the target procedure, unlike SPACER [103] or SMASH [82], which use two kinds of SMT queries.

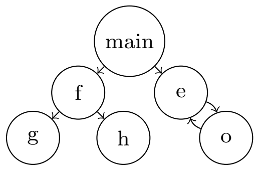
Table 4.1 lists the non-trivial verification steps that update various procedure summaries for the example. (Steps that do not update any summary are not listed.) The

```
main () {
  assert ¬(e(f() - 1));
}
f() {
  return h(g());
}
g() {
  x := havoc();
  return 2*x + 1;
}
h(x) {
  return x + 1;
}
e (x) {
  assume (x ≥ 0);
  if (x = 0)
    return true;
  else return o(x - 1);
}
o (x) {
  assume (x ≥ 1);
  if (x = 1)
    return true;
  return e(x - 1);
}
```

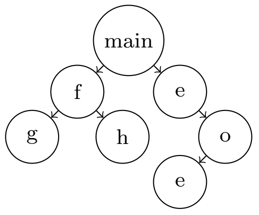
(a)

$$
\begin{array}{ll}
f(x) \land e(x-1,y) \land y & \Rightarrow \bot \\
g(x) \land h(x,y) & \Rightarrow f(y) \\
y = 2x+1 & \Rightarrow g(y) \\
y = x+1 & \Rightarrow h(x,y) \\
x \geq 0 \land x = 0 & \Rightarrow e(x,\top) \\
x \geq 0 \land x \neq 0 \land o(x-1,y) & \Rightarrow e(x,y) \\
x \geq 1 \land x = 1 & \Rightarrow o(x,\top) \\
x \geq 1 \land x \neq 1 \land e(x-1,y) & \Rightarrow o(x,y)
\end{array}
$$

(b)



(c)



(d)

Figure 4.1: Example: (a) source code, (b) CHC encoding, (c) call graph, and (d) final derivation tree.

Table 4.1: Relevant steps to verify program in Figure 4.1a.

	Call graph path	Environment	Target	Deductions (universally quantified)
1	main → e	Over	Under	$x = 0 \land y = \top \Rightarrow e(x,y)$
2	main → f → h	Over	Under	$y = x+1 \Rightarrow h(x,y)$
3	main → e → o	Over	Under	$x = 1 \land y = \top \Rightarrow o(x,y)$
4	main → f → g	Over	Under	$y \bmod 2 \neq 0 \Rightarrow g(y)$
5	main → f → g	Under	Over	$g(y) \Rightarrow y \bmod 2 \neq 0$
6	main → f → h	Under	Over	$h(x,y) \Rightarrow y = x+1$
7	main → f	Over	Under	$y \bmod 2 = 0 \Rightarrow f(y)$
8	main → f	Under	Over	$f(y) \Rightarrow y \bmod 2 = 0$
9	main → e → o → e	Over	Over	$(o(x,y) \Rightarrow y \Leftrightarrow ((1+x) \bmod 2 = 0)) \Rightarrow (e(m,n) \Rightarrow n \Leftrightarrow (m \bmod 2 = 0))$
10	main → e → o	Over	Over	$o(x,y) \Rightarrow y \Leftrightarrow ((1+x) \bmod 2 = 0))$ $e(x,y) \Rightarrow x > 1 \land y \Leftrightarrow ((1+x) \bmod 2 = 0)$

first column lists the call path that is visited in each step, in which the last call is the current *target* procedure whose summary is updated, and the call path is used to generate its bounded environment. The "Environment" (resp. "Target") column shows whether the bounded environment (resp. target) is over- or under-approximated. The "Deductions" column lists deductions resulting from SMT queries in that step; updates to over- or under-approximate summaries in O or U are made following these deductions. Note that formulas in this column (and in the remainder of this section) are implicitly universally quantified over all variables and involve uninterpreted predicates (e.g., $\mathtt{h}(\mathtt{x}, \mathtt{y})$ in row 2). Each uninterpreted predicate implicitly represents the actual semantics of the procedure that it encodes, where O and U are explicitly-maintained mappings to over- and under-approximations of these semantics. Except in row 9, all these formulas are implications that can be used to update the procedure summaries in O and U. Row 9 shows an implication between two such formulas – this is an instance of an EC lemma (described later in more detail), which is used to capture the relationship between mutually recursive procedures. The EC lemma on Row 9 expresses a relates the behaviors of the o and e procedure.

4.2 Using the Program Call Graph

The algorithm described in this chapter chooses environment-target pairs based on the call graph of the program, shown in Figure 4.1c. Note that this call graph's structure is also preserved in the CHCs in Figure 4.1b. For any uninterpreted predicate p that is the head of a CHC, for any uninterpreted predicate c in the body of the CHC, there is an edge from p to c in the call graph.

During exploration, the algorithm maintains explored paths through the call graph in a data structure called a *derivation tree*, initially consisting of only one node that represents the body of entry procedure `main`. Figure 4.1d shows a representation of the tree just before the algorithm converges. The subset A of nodes *available* to be explored is also maintained, and it is this subset that guides exploration in our algorithm.

Definition 4.2.1 (Derivation Tree)**.** A derivation tree $D = \langle N, E \rangle$ for a system of CHCs $\mathcal{C}$ is a tree with nodes N and edges E, where each $n \in N$ except the root node is labeled with uninterpreted predicate $pr(n)$, a *context* query CHC $ctx(n)$, and a *target application* $tgt(n)$ of $pr(n)$ within the body of $ctx(n)$. The root node r, which represents the `main` procedure, is such that $ctx(r) = Q$ for a query CHC $Q \in \mathcal{C}$ and $pr(n)$ and $tgt(n)$ both do not exist.

Every node in the derivation tree can be associated with the CHC that results from unfolding predicates according to the path from the root of the tree to the node, which is an *unbounded context* for the node. Here "unbounded" is used to distinguish contexts that have their sizes restricted by a given bound (called *bounded contexts* and introduced later) and those that are not (referred to as *unbounded*). Note that both bounded and unbounded contexts here are finite because paths through the tree are finite.

The CHCs for derivation tree nodes correspond to paths in the original program along which there may be potentially spurious counterexamples. In the example for the `main`-labeled node, this would be the query CHC $f(x) \wedge e(x-1, y) \wedge y \Rightarrow \perp$. For example, the node labeled `g` at the end of path `main` $\to$ `f` $\to$ `g` can be associated with the result of unfolding f in $f(x) \wedge e(x-1, y) \wedge y \Rightarrow \perp$ to get $g(w) \wedge h(w, x) \wedge e(x-1, y) \wedge y \Rightarrow \perp$. For this CHC to be valid under interpretations in mapping O, O needs to be such that replacing g, h, and e with their interpretations in O makes the body of the CHC unsatisfiable. Meanwhile, if the under-approximate mapping U is such that replacing g, h, and e with their interpretations in U makes the body of the CHC satisfiable, then there is a counterexample where the satisfying assignments for the variables determine the counterexample path through the original program.

As mentioned, each node has a target predicate application. This target application is the predicate application corresponding to the last call in the call path (in the case of the example, this would be the application of g so that $tgt(n) = g(w)$).

The disjunction of all possible conjunctions that could replace the predicate application if it were to be unfolded (in this case, $w = 2z + 1$) are referred to as the *body of the target*, since they constitute an encoding of the body of the target procedure in the original program. The unbounded environment for the node is the body of the associated unbounded context CHC before unfolding *without* the target predicate application, i.e. the environment for the target in the CHC, where an environment is defined as follows:

Definition 4.2.2 (Environment)**.** For a given CHC C of the form $R_1(\vec{x}_1) \wedge \ldots \wedge R_n(\vec{x}_n) \wedge \phi(\vec{x}) \Rightarrow \mathit{head}$, we say that the following formula is the *environment* (denoted $\mathit{ctx}(R_i, C)$) for the uninterpreted predicate application $R_i(\vec{x}_i)$:

$$R_1(\vec{x}_1) \wedge \ldots \wedge R_{i-1}(\vec{x}_{i-1}) \wedge R_{i+1}(\vec{x}_{i+1}) \wedge \ldots \wedge R_n(\vec{x}_n) \wedge \phi(\vec{x})$$

We naturally extend the mappings M from uninterpreted predicates to environments. That is, for the formula above: $M(\mathit{ctx}(R_i, C)) = \bigwedge\limits_{1 \leq j \leq n}^{j \neq i} M(R_j)(\vec{x}_j) \wedge \phi(\vec{x})$. Applying mappings to environments during verification allows us to use previously-learned procedure summaries, enabling modular verification.

In the case of considering the node labeled `g` in the example, the unbounded environment is $h(w, x) \wedge e(x - 1, y) \wedge y$. In general, a target predicate application's environment need not be taken from its unbounded context and can be, for example, a bounded environment as described in the next section.

For any node $n \in N$, let $\mathit{env}(n)$ refer to the environment for $\mathit{tgt}(n)$ in $\mathit{ctx}(n)$ and let $\mathit{body}(n)$ denote the body of the target $\mathit{tgt}(n)$.

Each node $n \in N$ represents a *verification subtask*, where the body of $\mathit{ctx}(n)$ represents a set of (potentially spurious) counterexamples. The goal of each subtask is to find a solution for the system of CHCs consisting of all non-query CHCs in $\mathcal{C}$ with the query CHC $\mathit{ctx}(n)$ and refine the over-approximation $O[\mathit{proc}(n)]$ to reflect the learned facts, or, if this cannot be done, to expand $\mathit{proc}(n)$'s under-approximation

$$\textbf{SAFE}\quad \frac{O \text{ is a solution for } \mathcal{C}}{D, A, O, U, L, \mathcal{C} \vdash \bot} \qquad\qquad \textbf{UNSAFE}\quad \frac{\mathit{body} \Rightarrow \bot \in \mathcal{C} \qquad U(\mathit{body}) \not\Rightarrow \bot}{D, A, O, U, L, \mathcal{C} \vdash \top}$$

Figure 4.2: Proof rules for concluding safety or unsafety as the result of the verification problem.

$U[proc(n)]$ to demonstrate (part of) a real counterexample. Exploration proceeds by choosing a node from A and making SMT queries about the body of the target and the environment for that node to try to learn formulas to update O and U with.

More formally, the algorithm generates and discharges a series of proof subgoals during execution, where a proof subgoal is defined as follows:

Definition 4.2.3 (Proof (sub)Goal)**.** For system of CHCs $\mathcal{C}$, derivation tree $D = \langle N, E \rangle$, a subset $A \subseteq N$ of *available* nodes, over- and under-approximate summary maps O and U, a set of EC lemmas L, and $\mathit{Res} \in \{\top, \bot\}$, a *proof (sub)subgoal* is denoted $D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}$.

The algorithm tries to prove either the case that $\mathit{Res} = \bot$ or that $\mathit{Res} = \top$. If it proves $\mathit{Res} = \bot$, it has verified P. If it proves $\mathit{Res} = \top$, then it has found a counterexample for P's safety. This proof proceeds with the application of proof rules, where the final rule applied is one of the two in Figure 4.2, where the **SAFE** rule corresponds to when the program is safe and the **UNSAFE** proof rule corresponds to when a counterexample is found. The remainder of the proof rules will be introduced and explained in the sequel.

4.3 Bounded Environments

Note that if unbounded contexts and environments are used as described, the longer the path in the call graph that we explore, the larger the SMT queries that we make will be. To improve scalability, the algorithm instead uses *bounded* contexts and environments from call paths to use in SMT queries at each step. These bounded environments include bodies of the ancestors of the target procedure, but only up to level k above the target in the call graph. Ancestors at $\ell > k$ above the target are soundly abstracted away so that these environments capture at least the behaviors of the program before and after the target procedure that may lead to a counterexample. Approximations of these environments and of the bodies of target procedures help us learn new facts about the targets.

Definition 4.3.1 (Bounded context)**.** For a given bound k, and a path $d = n_0 \rightarrow \ldots \rightarrow n_{m-1} \rightarrow n_m$ in a derivation tree, a *k-bounded context* for n_m is a formula $\mathit{bctx}(n_m)$ over free variables in $\mathit{unfold}(d,k)$, defined as follows:

$$\mathit{bctx}(n_m) \overset{\text{def}}{=} \mathit{unfold}(d,k).\mathit{body} \wedge \mathit{interface}(d,k) \wedge \mathit{summ}(d,k) \Rightarrow \perp$$

Here, we also have the following:

- $\mathit{unfold}(d,k)$ corresponds to unfolding the bodies of the last $k-1$ procedure calls in d into the body of $\mathit{proc}(n)$'s k^{th} ancestor in d, and $\mathit{unfold}(d,k).\mathit{body}$ denotes the body of this CHC
- $\mathit{interface}(d,k)$ is a formula over the inputs and outputs of the procedure for node n_{m-k}, $k < m$ (or $\top$, if $k \geq m$)
- $\mathit{summ}(d,k)$ is a formula over the inputs and outputs of the other callees of the k-bounded ancestors of $\mathit{proc}(n_m)$.

Note that $\mathit{unfold}(d,k)$ ignores any restrictions due to ancestors that are more than k-levels above $\mathit{proc}(n_m)$, and contains a subset of the conjuncts of the full context, which is given by $\mathit{unfold}(d,|d|) \Rightarrow \perp$. Such restrictions are expressed in $\mathit{interface}(d,k)$, which represents the interface between the k-bounded context and the rest of the unbounded context.

In practice, the algorithm computes $\mathit{interface}(d,k)$ as $\mathrm{QE}(\exists \mathit{free}.O(\mathit{env}(n_m)))$, where QE denotes quantifier elimination and $\mathit{free} \overset{\text{def}}{=} \mathit{fvs}(\mathit{env}(n_m)) \cup \mathit{fvs}(\mathit{unfold}(d,k))$, where *fvs* gives the set of free variables of a given formula. Quantifier elimination is approximated using the standard model-based projection technique [34]. Although it is always possible to use $\mathit{interface}(d, k) = \top$, which treats ancestor procedures above bound k as havocs, this choice was found to be ineffective in experiments.

In what follows, I refer to $\mathit{unfold}(d, k).\mathit{body} \wedge \mathit{interface}(d, k)$ as $B(d, k)$, or simply as B (when d and k are clear). Note that the algorithm requires that each $\mathit{bctx}(n_m)$ (and thus each $B(d, k)$) can be computed from its parent n_{m-1}'s bounded context via a single unfolding. Given the choice of $\mathit{interface}(d, k)$ mentioned earlier, using such a method to compute a child node's bounded context lets us avoid (approximate) quantifier elimination on large formulas since only one procedure body's variables need to be eliminated when starting from the parent's bounded context.

The $\mathit{summ}(d, k)$ formula can be either $\top$ or a conjunction that adds approximation constraints based on EC lemmas and summaries for the other callee procedures. The algorithm uses B as the body for *bctx* when $\mathit{summ}(d, k) = \top$, and otherwise uses $M(B) \wedge \mathit{tgt}(n)$ or $M(\mathit{env}(n)) \wedge \mathit{tgt}(n)$ as the body for *bctx*, where $M \in \{O, U\}$ (when $\mathit{summ}(d, k)$ is the conjunction from approximating with M).

This example uses $k = 2$. When the algorithm targets the last call to `e` along path $d =$ `main` $\rightarrow$ `e` $\rightarrow$ `o` $\rightarrow$ `e`, `main`'s body will be abstracted. Note that by the time this d is considered, the over-approximation for `f` is such that $O[f] = \lambda x.x \bmod 2 = 0$. For the corresponding node in the derivation tree, the bounded context is made of the following components:

- $\mathit{unfold}(d, 2) = x \geq 0 \wedge x \neq 0 \wedge x - 1 \geq 1 \wedge x - 1 \neq 1 \wedge e(x - 2, y)$

- $\mathit{interface}(d, 2) = x + 1 \bmod 2 = 0$ (following from $O[f]$)

- $\mathit{summ}(d, k) = \top$

4.4 Summary Updates using SMT Queries

There are four kinds of SMT queries that the algorithm uses to update summaries. Two over-approximate the body of the target and update the over-approximate summaries, and two under-approximate the body of the target and update under-approximate summaries. For each of the four SMT queries, the aim is to try to make deductions as seen in Table 4.1.

4.4.1 Updating over-approximate summaries

In checks that over-approximate the body of the target, the algorithm aims to learn an interpolant that proves the absence of a subset of counterexamples along the paths in the program that correspond to the chosen CHCs. That is, for a given node in the derivation tree, if we consider its corresponding CHC as described above, the algorithm tries to find an interpolant that separates the over-approximation of the target body from either an over-approximation of the environment or under-approximation of the environment, where over- (resp. under-)approximation refers to replacing all uninterpreted predicates with their interpretations in O (resp. U).

Since the target body is over-approximated, any interpolant found that separates it from the environment will be an over-approximate summary for the target procedure, expressing a fact about all behaviors of the procedure, and hopefully leading to the validity of the CHC corresponding to the given node in the derivation tree (though the algorithm does not check these CHCs for validity directly because they are not bounded in size). If validity is not achieved for the CHC for that node, then stronger over-approximations will be needed for one or more of the procedures whose predicates appear in the CHC (possibly including the target procedure for the node); despite not checking the full CHC for each node directly, the algorithm ensures that, without finding strong enough over-approximations that make the CHC valid, verification cannot succeed. Such over-approximate summaries allow us to prove safety in a modular way. For over-approximated environments, the existence of such an interpolant demonstrates the safety of the program along the relevant paths in the current node's CHC. For under-approximated environments, the existence of such an interpolant demonstrates the safety of the program only along *some* of those paths.

For all deductions in Table 4.1 that do not lead to EC lemmas and where the target body is over-approximated, the deductions are of the form $p(\vec{x}) \Rightarrow \mathbb{I}(\vec{x})$ for target procedure p and a formula $\mathbb{I}(\vec{x})$ with no uninterpreted predicates. $\mathbb{I}(\vec{x})$ is an interpolant that separates the over-approximation of the target body from the approximated environment. Whenever such a deduction is made, the over-approximate summary $O[p]$ is updated to yield a new over-approximate summary map O' where $O'[p]$ is as follows:

$$O'[p] = \lambda\vec{x}.O[p](\vec{x}) \wedge \mathbb{I}(\vec{x})$$

The two proof rules (and corresponding SMT checks) that corresponding to cases in which the over-approximate summaries are updated in this way are shown in Figure 4.3. The over-over (OO) rule involves over-approximating both the target procedure body and the environment and making an SMT query to find the interpolant $\mathbb{I}$. Meanwhile, the under-over (UO) rule is the same except the environment is under-approximated.

4.4.2 Updating under-approximate summaries

In checks that under-approximate the target body, the algorithm tries to find (part of) a bug. When the body of the target is under-approximated, the algorithm instead instead looks for cases where the conjunction of the under-approximated target body and approximated environment is satisfiable, where the satisfying assignment thus describes behaviors the procedure *may* exhibit. Under-approximate summaries

OVER-OVER (OO)

$$\frac{\begin{array}{c} n \in A \qquad p = \mathit{proc}(n) \\ O(\mathit{body}(n)) \Rightarrow \mathbb{I}(\vec{x}) \qquad \mathbb{I}(\vec{x}) \Rightarrow \neg O(\mathit{benv}(n)) \\ O' = O[p \mapsto \lambda \vec{x}.O[p](\vec{x}) \wedge \mathbb{I}(\vec{x})] \qquad D, A, O', U, L, \mathcal{C} \vdash \mathit{Res} \end{array}}{D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}}$$

UNDER-OVER (UO)

$$\frac{\begin{array}{c} n \in A \qquad p = \mathit{proc}(n) \\ U(\mathit{body}(n)) \Rightarrow \mathbb{I}(\vec{x}) \qquad \mathbb{I}(\vec{x}) \Rightarrow \neg O(\mathit{benv}(n)) \\ O' = O[p \mapsto \lambda \vec{x}.O[p](\vec{x}) \wedge \mathbb{I}(\vec{x})] \qquad D, A, O', U, L, \mathcal{C} \vdash \mathit{Res} \end{array}}{D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}}$$

Figure 4.3: The two proof rules that update over-approximate summaries.

allow us to construct counterexamples in the case where the program is unsafe. Note that for under-approximated *unbounded* environments, the existence of such a satisfying assignment always demonstrates an actual counterexample in the program. For bounded environments or over-approximated environments, the sound abstraction/approximation in the environment allows for more behaviors than are in the original program, so a satisfying assignment may correspond to a counterexample that is spurious in the actual program. In both over- and under-approximate cases, approximating and bounding the environment and approximating the target body allows us to keep queries small.

For all deductions in Table 4.1 where target body is under-approximated, the deductions for a target procedure p can be seen to be of the form $F(\vec{x}) \Rightarrow p(\vec{x})$ for a formula F without uninterpreted predicates. Whenever such a deduction is made, the under-approximate summary $U[p]$ is updated to a new under-approximate mapping U' as follows:

$$U'[p] \leftarrow \lambda \vec{x}.U[p](\vec{x}) \vee F(\vec{x})$$

Note that each such F found for a node n in the derivation tree must be such that

$$F(\vec{x}) \Rightarrow \exists \mathit{fvs}(\mathit{body}(n)) \setminus \vec{x}.U(\mathit{body}(n))$$

UNDER-UNDER (UU)

$$\frac{\begin{array}{c} n \in A \qquad p = \mathit{proc}(n) \\ U(\mathit{body}(n)) \wedge U(\mathit{benv}(n)) \not\Rightarrow \bot \\ F(\vec{x}) \Rightarrow \exists \mathit{locals}(n).U(\mathit{body}(n)) \\ U' = U[p \mapsto \lambda\vec{x}.U[p](\vec{x}) \vee F(\vec{x})] \qquad D, A, O, U', L, \mathcal{C} \vdash \mathit{Res} \end{array}}{D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}}$$

OVER-UNDER (OU)

$$\frac{\begin{array}{c} n \in A \qquad p = \mathit{proc}(n) \\ U(\mathit{body}(n)) \wedge O(\mathit{benv}(n)) \not\Rightarrow \bot \\ F(\vec{x}) \Rightarrow \exists \mathit{locals}(n).U(\mathit{body}(n)) \\ U' = U[p \mapsto \lambda\vec{x}.U[p](\vec{x}) \vee F(\vec{x})] \qquad D, A, O, U', L, \mathcal{C} \vdash \mathit{Res} \end{array}}{D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}}$$

Figure 4.4: The two proof rules that update under-approximate summaries.

where $\mathit{fvs}(\mathit{body}(n))$ gives all the free variables in $\mathit{body}(n)$. Such an $F(\vec{x})$ can be found by either using the consequent of the implication as $F(\vec{x})$ or by employing methods for under-approximating of quantifier elimination such as the model-based projection technique [34]. Using such an F to perform the update weakens $U[p]$ while preserving the invariant that $\forall\vec{x}.U[p](\vec{x}) \Rightarrow O[p](\vec{x})$ (recall this invariant for under-approximate mappings from Chapter 2); assuming that the invariant already holds, then $U'[p] \Rightarrow U(\mathit{body}(n)) \Rightarrow O(\mathit{body}(n)) \Rightarrow O[p](\vec{x})$.

Let $\mathit{locals}(n) = \mathit{fvs}(\mathit{body}(n)) \setminus \vec{x}$ where $\mathit{tgt}(n) = p(\vec{x})$. The two proof rules (and corresponding SMT checks) that correspond to cases in which the under-approximate summaries are updated in this way are shown in Figure 4.4. The under-under (UU) rule involves under-approximating both the target procedure body and the environment and making an SMT query to find the formula $F(\vec{x})$. Meanwhile, the over-under (OU) rule is the same except the environment is over-approximated.

4.4.3 Example

Let us now consider the SMT queries on a chosen environment-target pair at each step. Suppose our algorithm has already considered the path to o on row 3 (Table 4.1) and now chooses node `g` $\in A$ in path `main` $\rightarrow$ `f` $\rightarrow$ `g`. Recall that the unbounded environment here is $h(w, x) \wedge e(x-1, y) \wedge y$. For the bound $k = 2$, this unbounded environment is also the bounded environment. Since the bounded environment includes predicate applications corresponding to calls to `h` (called by `f`) and `e` (called by `main`), we use their over-approximate summaries (both currently $\top$) in SMT queries requiring over-approximation. We similarly under-approximate using summaries for `h` and `e` learned in rows 2 and 1, respectively. Over- and under-approximations of `g` are just the body of its CHC with any variables (i.e., "CHC-local variables") that do not occur in the head of the CHC rewritten away (i.e., y mod $2 \neq 0$[1]), since it has no callees.

The *over-over* query that over-approximates both the environment and target body fails; the conjunction of the approximated environment and target body is satisfiable. Similarly, the *under-under* query that under-approximates both the environment and target body fails, since the conjunction of the approximated environment and target body is unsatisfiable.

A weaker version of the *under-under* check is the *over-under* check, in which the environment is now *over-approximated.* Because it is weaker, it may result in learning under-approximate summaries that may not be necessary, since the over-approximated environment may contain spurious counterexamples. When our algorithm performs this check, it finds a path that goes through the over-approximated environment and the under-approximation of `g`'s body and thus augments `g`'s under-approximation (row 4).

A corresponding weaker version of the *over-over* check is the *under-over* check, in which the environment is *under-approximated.* Because the under-approximated environment may not capture all counterexamples, the learned interpolant by itself

[1]Expressions such as y mod $2 = 0$ can be generated by existentially quantifying local variables and then performing quantifier elimination.

could be too weak to prove safety. Our algorithm refines `g`'s over-approximation with the interpolant learned in this query (row 5).

Note that these two weaker checks are crucial in our algorithm. Consider a different `main` function that contains only `assert(f() mod 2 = 0)`. To prove safety, we would need to consider paths `main` $\rightarrow$ `f` $\rightarrow$ `h` and `main` $\rightarrow$ `f` $\rightarrow$ `g`, but for these paths, both "stronger" checks fail. Paths through the derivation tree must be paths through the call graph, so we would not consider the bodies of `h` and `g` simultaneously; the "weaker" checks allow us to learn summary updates.

4.5 Explicit Induction and EC Lemmas

So far, we have not considered nor seen how to use EC lemmas for discharging verification subproblems; however, they are necessary for enabling the algorithm to handle mutual recursion. To demonstrate the need for and use of induction and EC lemmas for handling mutual recursion, we now consider row 9 in Table 4.1, where we perform an over-over check for the final call to `e` in the call path. The current derivation tree has the same structure as the final derivation tree, shown in Figure 4.1d.

No induction. At this stage, our over-approximation for `f` precisely describes all possible behaviors of `f` (rows 7, 8), but no interpolant can be learned because the over-approximation $\top$ of `o` in the body of the target `e` is too coarse. Without using induction, we cannot make any assumptions about this call to `o`, and are stuck with this coarse over-approximation. Even if we inlined `o` in `e`, we would similarly still have an overly coarse over-approximation for `e`.

Induction without EC lemmas. We can instead try to use induction on the body of `e`. Its over-approximated environment includes counterexample paths that we would like to prove spurious. To do this, we augment the OO rule from Figure 4.3 as shown in Figure 4.5. This OOI rule has two changes from the OO rule: the first

$$\textbf{OVER-OVER-IND}\ (\mathrm{OOI})\quad \frac{\begin{array}{c} n \in A \qquad p = \mathit{proc}(n) \qquad \mathit{hyp} = \forall \vec{x}.O[p](\vec{x}) \Rightarrow \mathbb{I}(\vec{x}) \\ O(\mathit{body}(n)) \land \mathit{hyp} \Rightarrow \mathbb{I}(\vec{x}) \qquad \mathbb{I}(\vec{x}) \Rightarrow \neg O(\mathit{benv}(n)) \\ O' = O[p \mapsto \lambda \vec{x}.O[p](\vec{x}) \land \mathbb{I}(\vec{x})] \qquad D, A, O', U, L, \mathcal{C} \vdash \mathit{Res} \end{array}}{D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}}$$

Figure 4.5: Altered OO rule from Figure 4.3 with induction.

is the additional premise $\mathit{hyp} = \forall \vec{x}.O[p](\vec{x}) \Rightarrow \mathbb{I}(\vec{x})$, which denotes the induction hypothesis, and the second is modifying the implication $O(\mathit{body}(n)) \Rightarrow \mathbb{I}(\vec{x})$ so that the induction hypothesis is now a conjunct in the antecedent. This implication checks both the base case and the inductive step.

The base case for the induction is that $\mathbb{I}$ over-approximates $\mathit{proc}(n)$ for all CHCs that do not have applications of $\mathit{proc}(n)$ in their body. For the inductive step, we consider such CHCs where $\mathit{body}(n)$ contains applications of $\mathit{proc}(n)$. The induction hypothesis, which is captured by formula hyp, is that $\mathbb{I}$ over-approximates all recursive calls to $\mathit{proc}(n)$ inside these bodies.

Let us now consider applying this rule to our example. Let formula hyp denote the property $\forall x, y.O(\phi(x, y))$, where $\phi(x, y) \overset{\text{def}}{=} e(x, y) \Rightarrow (y \Leftrightarrow x \bmod 2 = 0)$. The consequent in the implication is generated by examining the current environment for the target application of e, i.e., the environment implies the negation of the consequent. Problems arise when trying to prove this property by induction because there is no opportunity to apply the induction hypothesis about e.

More specifically, $O(\mathit{body}(n))$ contains a disjunct corresponding to the case in the else branch of the original program in which `e` calls `o`. This disjunct is $x \geq 0 \land x \neq 0 \land o(x - 1, y)$. Facts about o are thus needed in O to finish the proof for $O(\mathit{body}(n)) \land \mathit{hyp} \Rightarrow \mathbb{I}(\vec{x})$ because $O(\mathit{body}(n))$ involves using summary $O[o]$, and with a coarse summary like $O[o] = \lambda \vec{x}.\top$, the implication cannot be proven.

If we were to inline `o` in `e` and assume the induction hypothesis that $\phi(x, y)$ holds for the inner call to `e`, an inductive proof would succeed without using EC lemmas. However, such an inlining approach can lead to poor scalability and precludes inference of summaries (e.g., for `o`) that could be useful in other call paths.

EC lemmas. The algorithm discovers additional lemmas in the form of implications over certain procedure summaries. In order to use these lemmas, the algorithm employs an altered version OOIL of the OOIL rule from Figure 4.5, as well as a version UOIL where the environment is under-approximated, as shown in Figure 4.6.

These rules make assumptions for current node n and perform induction using these assumptions and known EC lemmas in L. In particular, $\mathit{assumps}(n, D)$ is a set of assumptions $\{a_i \mid 1 \leq i \leq j\}$ for some $j \geq 0$. When $j = 0$, the set of assumptions is empty, and OOIL has the same effect as applying OOI, except that the learned information is stored as an EC lemma rather than as part of O (applications of the EL and RL rules after OOIL can achieve the same effect as the OOI rule). Each assumption a_i is of the following form:

$$q_i(\vec{x}_i) \Rightarrow \forall \mathit{fvs}(b_i) \setminus \vec{x}_i . \neg b_i,$$

where q_i is the predicate for an ancestor of n and q_i is called by target p in some CHC. The ancestor node's bounded environment is b_i. Intuitively, each assumption is that the ancestor's bounded verification subtask has been discharged.

The *Inst* function takes a set of assumptions S, adds a conjunct $a_i[\vec{x}_i \mapsto \vec{x}]$ for each predicate application $q_i(\vec{x})$ in $\mathit{body}(n)$ to each each element $a_i \in S$, conjoins the resulting formulas, and replaces each application of an uninterpreted predicate with its interpretation in O.

Compared to the OOI rule, the OOIL has three differences: the first is the making of assumptions as can be seen in the addition of premise $S = \mathit{assumps}(n, D)$, the second is that the subgoal has an update to L rather than O, and the third is the use of these premises and EC lemmas as can be seen by the addition of conjuncts $\bigwedge O(L)$ and $\mathit{Inst}(S)$ to the antecedent of implication $O(\mathit{body}(n)) \wedge \mathit{hyp} \Rightarrow \mathbb{I}(\vec{x})$. As in the OOI case, this implication checks both the base case and inductive step, but now under the assumptions made in S.

When EC lemmas are learned with $j = 0$ (i.e., S is empty), then these degenerate EC lemmas are essentially the same as over-approximate summaries. The EL rule in 4.6 allows such lemmas to be converted into over-approximate summaries in O, allowing the combination of OOIL (resp. UOIL) and EL to replace the usage of OO (resp. UO). As over-approximate summaries are updated, EC lemmas can also be weakened, since they may have assumptions that become accounted for in the updated over-approximate summaries. The RL rule allows EC lemmas to be weakened when this is the case, and can eventually lead to the EL rule being able to be applied for a sufficiently weakened EC lemma.

Let us now consider applying the OOIL rule to our example. Let hyp be $\forall x, y.\phi(x, y)$ as before. Note that $\mathbb{I}(m, n)$ is thus $n \Leftrightarrow m \bmod 2 = 0$. Let $\theta(m, n) \overset{\text{def}}{=} o(m, n) \Rightarrow (n \Leftrightarrow (1 + m) \bmod 2 = 0)$. (As with $\phi(x, y)$, the consequent in this implication is generated by examining the environment for the target application of o.) The set of assumptions S contains only $\forall m, n.\theta(m, n)$, since $j = 1$ is used for computing S in this case[2].

We now can use the assumption about the evenness of the result of e (as captured by $\phi(x, y)$) to help prove by induction that o's output is always odd (as captured by hyp). This part of the proof is captured by the following premise:

$$O(\mathit{body}(n)) \wedge \bigwedge O(L) \wedge \mathit{Inst}(S) \wedge \mathit{hyp} \Rightarrow \mathbb{I}(x - 2, y)$$

Note here that $\bigwedge O(L) = \top$ because $L = \varnothing$ and that $\mathit{Inst}(S) \overset{\text{def}}{=} \theta(x - 3, y)$ as a conjunct. Here, $O(\mathit{body}(n)) = (x - 2 \geq 0 \wedge x - 2 = 0 \wedge y = \top) \vee (x - 2 \geq 0 \wedge x - 2 \neq 0 \wedge O[o](x - 3, y))$.

[2]Note that this choice of $j = 1$ is arbitrary, but it is preferable to use the smallest j possible to learn stronger EC lemmas

OVER-OVER-IND-LEMMAS (OOIL)

$$\frac{\begin{array}{c} n \in A \qquad p = \mathit{proc}(n) \qquad \mathit{hyp} = \forall \vec{x}.O[p](\vec{x}) \Rightarrow \mathbb{I}(\vec{x}) \qquad S = \mathit{assumps}(n, D) \\ O(\mathit{body}(n)) \wedge \bigwedge O(L) \wedge \mathit{Inst}(S) \wedge \mathit{hyp} \Rightarrow \mathbb{I}(\vec{x}) \qquad \mathbb{I}(\vec{x}) \Rightarrow \neg O(\mathit{benv}(n)) \\ L' = L \cup \{\forall \mathit{fvs}(S), \vec{x}. \bigwedge S \Rightarrow (p(\vec{x}) \Rightarrow \mathbb{I}(\vec{x}))\} \qquad D, A, O, U, L', \mathcal{C} \vdash \mathit{Res} \end{array}}{D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}}$$

UNDER-OVER-IND-LEMMAS (UOIL)

$$\frac{\begin{array}{c} n \in A \qquad p = \mathit{proc}(n) \qquad \mathit{hyp} = \forall \vec{x}.O[p](\vec{x}) \Rightarrow \mathbb{I}(\vec{x}) \qquad S = \mathit{assumps}(n, D) \\ O(\mathit{body}(n)) \wedge \bigwedge O(L) \wedge \mathit{Inst}(S) \wedge \mathit{hyp} \Rightarrow \mathbb{I}(\vec{x}) \qquad \mathbb{I}(\vec{x}) \Rightarrow \neg U(\mathit{benv}(n)) \\ L' = L \cup \{\forall \mathit{fvs}(S), \vec{x}. \bigwedge S \Rightarrow (p(\vec{x}) \Rightarrow \mathbb{I}(\vec{x}))\} \qquad D, A, O, U, L', \mathcal{C} \vdash \mathit{Res} \end{array}}{D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}}$$

REDUCE-LEMMAS (RL)

$$\frac{\begin{array}{c} ec = \forall \mathit{vars}(S), \vec{x}. \bigwedge S \Rightarrow (p(\vec{x}) \Rightarrow \mathit{prop}) \in L \\ a \in S \qquad (p'(\vec{x}) \Rightarrow O[p'](\vec{x})) \Rightarrow a \qquad S' = S \setminus \{a\} \\ L' = (L \setminus \{ec\}) \cup \big\{\forall \mathit{vars}(S'), \mathit{in}, \mathit{out}. \bigwedge S' \Rightarrow \big(p(\mathit{in}, \mathit{out}) \Rightarrow \psi\big)\big\} \\ D, A, O, U, L', \mathcal{C} \vdash \mathit{Res} \end{array}}{D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}}$$

ELIM-LEMMAS (EL)

$$\frac{\begin{array}{c} ec = \forall \vec{x}.\top \Rightarrow (p(\vec{x}) \Rightarrow \mathit{prop}) \in L \\ O' = O[p \mapsto \lambda \vec{x}.O[p](\vec{x}) \wedge \mathit{prop}] \\ D, A, O', U, L \setminus \{ec\}, \mathcal{C} \vdash \mathit{Res} \end{array}}{D, A, O, U, L, \mathcal{C} \vdash \mathit{Res}}$$

Figure 4.6: Proof rules for induction with EC lemmas.

The first disjunct in $O(\mathit{body}(n))$ represents the base case. Let us consider the case where the first disjunct is true. To prove the premise, it is sufficient to show that the following holds:

$$x - 2 \geq 0 \wedge x - 2 = 0 \wedge y = \top \Rightarrow (y \Leftrightarrow x - 2 \bmod 2 = 0)$$

From the conjuncts $x - 2 = 0$ and $y = \top$, it clearly holds.

Let us now consider the case where the second disjunct is true, which corresponds to the inductive step. To prove the premise, it is sufficient to show that the following holds:

$$x - 2 \geq 0 \land x - 2 \neq 0 \land O[o](x-3, y) \land \theta(x-3, y) \Rightarrow (y \Leftrightarrow x - 2 \bmod 2 = 0)$$

From the conjunct $\theta(x-3, y)$, which equals $y \Leftrightarrow x - 2 \bmod 2 = 0$, this implication also clearly holds.

The other premises of the OOIL rule are easy to prove in this case, and, as a result, the formula $\mathbb{I}$ is learned as part of an *EC lemma* and added to the set of EC lemmas L to yield the new set L'. See row 9 of Table 4.1 for the corresponding deduction, which states, essentially, that given that the result of `o` is `true` iff its input is odd, the result of `e` is `true` iff its input is even.

Now, let us reconsider the call to `o` along call path `main` $\rightarrow$ `e` $\rightarrow$ `o`. The discovered EC lemma allows us to prove formula θ valid by induction and learn it as part of an over-approximate summary by using the OOIL and then EL rules. This new over-approximate fact for o is combined with the EC lemma in the RL rule allowing the algorithm to use the EL rule to learn the deduction $e(x, y) \Rightarrow (y \Leftrightarrow x \bmod 2 = 0)$, which it uses as part of an update of O (from row 10 in Figure 4.1).

At this point, the over-approximate summaries in O are sufficient to prove safety; the **SAFE** rule from Figure 4.2 can be applied.

4.6 Modular Verification Algorithm

This section describes more precisely how the modular verification algorithm employs the proof rules described above. I first outline the top-level procedure (Sect. 4.6.1) based on iteratively processing nodes in the derivation tree. I then describe the order in which SMT queries are performed (Sect. 4.6.2), and the heuristics I have found useful in practice (Sect. 4.7). I then present the correctness and the progress property of the algorithm and discuss limitations (Sect. 4.6.4).

Algorithm 4 Procedure for solving multiple queries

```
1: procedure SOLVE(CHCs $\mathcal{C}$)
2:     for $R \in \mathcal{R}$ do $O[R] \leftarrow \lambda x_1, \ldots, x_n.\top$
3:     for $R \in \mathcal{R}$ do $U[R] \leftarrow \lambda x_1, \ldots, x_n.\perp$
4:     $Queries \leftarrow$ GETQUERIES($\mathcal{C}$)
5:     $\mathcal{C} \leftarrow \mathcal{C} \setminus Queries$
6:     for $Q \in Queries$ do
7:         $\mathcal{C} \leftarrow \mathcal{C} \cup \{Q\}$
8:         $(result, Goal) \leftarrow$ VERIFY($\mathcal{C}, O, U, Q$)
9:         if $result \neq safe$ then
10:            return $result$
11:        $O \leftarrow O$ in $Goal$
12:        $U \leftarrow U$ in $Goal$
```

4.6.1 Algorithm Outline

The algorithm constructs a derivation tree based on the call graph of the program, which is used to guide the selection of CHCs to explore. It achieves scalability by considering only bounded environments in all the SMT queries, as described previously. Recall that these SMT queries are implicitly made during the application of proof rules. The use of induction and EC lemmas enables handling of *mutually recursive programs*. The state during verification is captured by proof (sub)goals as defined earlier.

Main loop. For systems of CHCs $\mathcal{C}$ with queries $Q_0, \ldots, Q_n$, the full system of CHCs can be solved by invoking the SOLVE procedure shown in Algorithm 4. This SOLVE procedure assumes that any updates made to mappings O and U in the VERIFY procedure also affect the O and U mappings within SOLVE.

The VERIFY procedure shown in Algorithm 5 accepts a system of CHCs $\mathcal{C}$, over- and under-approximate summary mappings O and U, and a query CHC Q. Note that Q should be the only query CHC in $\mathcal{C}$. It first constructs an initial proof goal containing an initial derivation tree, initial summary maps, and empty sets of lemmas. Initially all nodes in the derivation tree are available, i.e., they are in A. It

Algorithm 5 Modular verification procedure for a single query

```
1: procedure VERIFY(CHCs $\mathcal{C}$ with uninterpreted predicates $\mathcal{R}$, $O$, $U$, $Q \in \mathcal{C}$)
2:     $N \leftarrow \{r\}$ with $ctx(r) = Q$
3:     $D \leftarrow \langle N, \varnothing \rangle$
4:     for predicate application $R(\vec{x})$ in $Q$ do
5:         $D.E \leftarrow D.E \cup \{r \rightarrow n\}$ where $tgt(n) = R(\vec{x})$ and $ctx(n) = Q$
6:     $Goal \leftarrow D, N \setminus \{r\}, O, U, \varnothing, \mathcal{C} \vdash Res$
7:     while $Goal.A \neq \varnothing$ or summaries are insufficient do
8:         $Goal \leftarrow$ PROCESSNODE($n$, $Goal$) for $n \in Goal.A$
9:     return (RESULT($Goal$), $Goal$)
```

then iteratively chooses an available node and tries to update its summaries (using routine ProcessNode), thereby updating the current goal. The loop terminates when no more nodes are available or when the current summaries are sufficient to prove/disprove safety. Result returns *safe* if the summaries are sufficient for proving safety, *unsafe* if they are sufficient for disproving safety, or *unknown* otherwise.

Throughout the algorithm, proof rules are applied, as will be demonstrated in subsequent sections; the current *Goal* is updated whenever a proof rule is applied. Note that the algorithm is building a proof tree from the bottom-up, so an application of a rule here involves matching the conclusion to the current *Goal*.

Choice of procedures and environments. ProcessNode can be viewed as making queries on an environment-procedure pair. If the algorithm chooses node n, then the pair consists of $benv(n)$ and the procedure corresponding to $proc(n)$. Note that the call graph guides the choice of the target since all paths in D correspond to call graph paths, and the bounded environment, which is computed by unfolding the k-bounded ancestors of the target. Importantly, the chosen node must be in A; this choice can be heuristic as long as no node in A is starved. Handling nodes in A in a first-in-first-out manner is sufficient.

Algorithm 6 Procedure to learn from a node.

```
1: procedure ProcessNode($n$, $Goal$)
2:     if OU($n$, $Goal$) then UU($n$, $Goal$)
3:     if no UU call above returned $true$ then
4:         if $\neg$OOIL($n$, $Goal$) then UOIL($n$, $Goal$)
5:     if no UU nor OOIL call above returned $true$ then AddNodes($n$, $Goal$)
6:     $updated \leftarrow$ any summaries were updated above
7:     Processed($n$, $updated$, $Goal$)
8:     return $Goal$
```

Summary Inference. The algorithm learns new summaries for target predicates by applying the four proof rules outlined earlier (OOIL, UU, OU, UOIL). Note that by choosing the set of assumptions and inductive hypotheses properly, the OOIL and UOIL rules can achieve the same effect as applying the non-inductive OO and UO rules.

4.6.2 Ordering and Conditions for SMT Queries

The way in which proof rules are applied to process a node is shown in Algorithm 6. In the pseudocode, OOIL, UOIL, OU, and UU refer to attempts to apply the corresponding rules, where the OOIL, UOIL calls always follow the application of either the OOIL and UOIL rule by as many applications of the RL and EL rules as possible. They return *true* upon successful application (and update *Goal*), or *false* otherwise.

If the algorithm has neither found any counterexamples through the bounded environment (i.e., all UU attempts failed), nor eliminated the bounded verification subtask (i.e., the OOIL attempt failed), then it tries to derive new facts by adding new available nodes for the callees of $proc(n)$. Procedure AddNodes adds these nodes while avoiding adding redundant nodes to D. If any summary updates were made for $proc(n)$, then the procedure Processed (line 7) will add the bounded parents of n to A, so that new information can be propagated to the parents' summaries. It then removes n from A.

ADD-NODE (AN)

$$\frac{n \in A \qquad tgt(n') = p(\vec{x}) \qquad bctx(n') = benv(n) \wedge body(n) \qquad \text{tgt}(n') \text{ in } body(n) \qquad D'.E = D.E \cup \{n \to n'\} \qquad D', A, O, U, L, \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

MAKE-AVAILABLE (MA)

$$\frac{n \in D.N \qquad tgt(n) = p(\vec{x}) \qquad \forall n' \in A.tgt(n') = p(\vec{x'}) \Rightarrow \forall \vec{x}.benv(n') \neq \forall \vec{x'}.benv(n) \qquad A' = A \cup \{n'\} \qquad D', A, O, U, L, \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

MAKE-UNAVAILABLE (MU)

$$\frac{n \in A \qquad D, A \setminus \{n\}, O, U, L, \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

Figure 4.7: Proof rules for adding and removing nodes from A.

Algorithm 7 Procedure for adding nodes in derivation tree

```
procedure AddNodes($n$, $Goal$)
    for $body \Rightarrow proc(n)(\vec{x}) \in \mathcal{C}$ do
        for predicate application $p(\vec{x})$ in $body$ do
            $n' \leftarrow$ AddNode($n$, $p(\vec{x})$, $Goal$)
            MakeAvailableIfNew($n'$, $Goal$)
```

4.6.3 Addition and Removal of Nodes in Derivation Tree

Additional proof rules specify the removal and addition of nodes in D and A. These can be found in 4.7 and consist of the Add-Node (AN) rule to add new nodes to the derivation tree D, the Make-Available (MA) rule to add an existing and non-redundant node to the available set A, and the Make-Unavailable (MU) rule to remove a node from A. These proof rules are used by the AddNodes procedure shown in Algorithm 7 as well as the Processed procedure.

Adding nodes. For every CHC in $\mathcal{C}$ whose head is an application of $proc(n)$, the AddNodes calls procedure AddNode($n, p(\vec{x}), Goal$), which applies AN to $Goal$ with premises matching its arguments, adding a new node n' to the derivation tree in the subgoal. After AddNode applies AN, then it updates $Goal$ to be the subgoal of the application. After, the AddNodes procedure calls MakeAvailableIfNew to make the just-added node n' available in A if it would not be redundant to do so. MakeAvailableIfNew, tries to apply MA if either of the following hold:

- n' has never been processed before
- n' has previously been processed with summaries O_{prev} and U_{prev} and bounded environment $benv(n')$ has a different over- or under-approximation than before, i.e., $M_{prev}(body(n')) \neq M(body(n'))$ or else $M_{prev}(benv(n')) \neq M(benv(n'))$ for $M \in \{O, U\}$

Similarly to AddNode, the procedure MakeAvailableIfNew$(n', p(\vec{x}), Goal)$ applies MA with premises that match its arguments. As with AddNode, MakeAvailableIfNew has the side-effect of updating *Goal* to be the subgoal of the applied rule if the rule is successfully applied.

If MakeAddNode fails, then there is already a node n'' in A with the same target predicate and bounded environment that the node n' would have. I.e., there is another node $n'' \in A$ with a target predicate application $tgt(n'')$ and bounded environment $benv(n'')$ that are equivalent modulo renaming to $tgt(n')$ and $benv(n')$. Adding a new node to A for this target predicate and bounded environment would be redundant, since the SMT queries would be the same (modulo renaming) as for the node n', so n is not added. For a node $n \in D.N$, a node $n' \in D.N$ such that its predicate and bounded environment are the same as n's in this way is called an *equivalent node*. Note that all nodes n are equivalent nodes to themselves.

Removing nodes. The Processed procedure called by ProcessNode is shown in Algorithm 8. It is such that if its argument *updated* is *true*, it applies the MU rule on its argument n and updates *Goal* to the subgoal in this application. It also makes certain nodes available. These nodes are those that have a verification subproblem represented by n or an equivalent node, meaning that they are a subset of those that,

Algorithm 8 Procedure for updating available nodes in derivation tree.

```
procedure PROCESSED($n$, $updated$, $Goal$)
    if $updated$ then
        MAKEUNAVAILABLE($n$, $Goal$)
        for equivalent nodes $n'$ to $n$ in $D.N$ do
            $n''$ is the parent of $n'$ in $D.E$
            MAKEAVAILABLEIFNEW($n''$, $Goal$)
```

upon being processed again with the current O and U maps, would have different queries made about them because of the update to $M[proc](n)$ for some $M \in \{O, U\}$. In the case that, *updated* is *false*, PROCESSED does nothing.

4.6.4 Correctness and Progress

The correctness and progress claims for Algorithm 5 are stated below.

Theorem 4.6.1 (Correctness)**.** *Algorithm 5 returns safe (resp. unsafe) only if the program with entry point* `main` *never violates the assertion (resp. a path in* `main` *violates the assertion).*

Proof. The CHC encoding is such that there is a solution to the system of CHCs $\mathcal{C}$ iff the program does not violate the assertion. As a result, if the over-approximate summaries O constitute a solution and proof rule SAFE can be applied, the program does not violate the assertion. The under-approximate summaries U in every proof subgoal are guaranteed to be such that for any $p \in \mathcal{C}$, $U[p]$ implies any over-approximation $O[p]$. If UNSAFE can be applied, then the under-approximate summaries U imply that there is no possible solution O. The summaries in U can be used to reconstruct a counterexample path through the original program in this case. □

Theorem 4.6.2 (Progress)**.** *Processing a node in the derivation tree leads to at least one new (non-redundant) query.*

Proof. Initially, no nodes in A have been processed, and after a node is processed, it is removed from the derivation tree. The only way that a node can be processed and not have a new query made about it is if an already-processed node is re-added to A and this node does not have a new query that can be made about it. The MakeAvailableIfNew procedure is the only one that adds nodes to A and, by definition, will only add a node to A if there is a new query that can be made about it. □

Limitations. If the underlying solver is unable to find appropriate interpolants, the algorithm may generate new queries indefinitely. (The underlying problem is undecidable, so this is not unusual for modular verifiers.) Note, however, that because environments are bounded, each query's size is restricted.

4.7 Heuristics

In this section, I will discuss certain heuristics that can be and are used in the implementation of the algorithm.

4.7.1 Prioritizing choice of node

The Verify procedure from Figure 5 employs a heuristic to choose which node in the set A to call ProcessNode on next. The factors that contribute toward a node's priority are as follows, with ties in one factor being broken by the next factor, where $\mathit{depth}(n)$ denotes the depth of node n in D and $\mathit{previous}(n)$ denotes the number of times that the node n has been chosen previously:

- A lower $\alpha * \mathit{depth}(n) + \beta * \mathit{previous}(n)$ score gives higher priority, where α and β are weights
- A lower call graph depth of $\mathit{proc}(n)$ gives higher priority
- A predicate application $\mathit{tgt}(n)$ that syntactically occurs later in the CHC gives higher priority

The implementation prioritizes nodes n with lower $\mathit{depth}(n)$ values because they are more likely to help propagate learned summaries up to the *main* procedure's callees. This priority is moderated by the $\mathit{previous}(n)$ score which should prevent the starvation of nodes with larger $\mathit{depth}(n)$ values. The current heuristic search is more BFS-like, but for some examples, a DFS-like search is better.

4.7.2 Avoiding Redundant Queries

If the algorithm has previously considered a node n that it is now processing, it can avoid making the same queries that we have previously made; while it is guaranteed that at least one of the four SMT queries will be new, not all of them necessarily will be. For example, if none of the over-approximate summaries for any of the predicates in $\mathit{benv}(n)$ nor any of over-approximate summaries for any of the procedures called by $\mathit{proc}(n)$ have been updated since the last time n was processed, the algorithm does not need to redo the over-over check. The implementation keeps track of whether or not there have been relevant changes in O and U since the last time queries for a node n were performed in order to make sure redundant queries are not issued.

4.7.3 Learning Over-approximate Bodies

Although there are many existing methods to interpolate, in many cases they may not be useful (e.g., if an interpolant is just $\top$, it will not yield any semantic refinement to an over-approximate summary). To improve the chances of learning a refinement for an over-approximate summary, whenever the algorithm applies one of the proof rules that involves over-approximating the procedure body (e.g., OOIL, UOIL), it ensures at least that it learns the result of over-approximating the procedure body as an over-approximate fact. For example, consider doing this for the rule OOIL. The algorithm would simply replace premise $D, O, U, L', P \vdash \mathit{Res}$ with $D, O[p \mapsto \lambda\vec{x}.O[p](\vec{x}) \wedge \exists \mathit{locals}(n).O(\mathit{body}(n))], U, L', P \vdash \mathit{Res}$. Note that the result of applying over-approximate quantifier elimination to $O[p](\vec{x}) \wedge \exists locals.O(body(n))$ can still yield a reasonably good update.

4.7.4 Preventing summaries from growing too large

Although it is desirable to increase chances of learning useful refinements of over-approximations as just discussed, it is still also desirable to prevent summaries from becoming too complicated. This can be achieved in a few ways.

Quantifier Elimination One way is to use quantifier elimination or an approximation thereof on each conjunct (resp. disjunct) that is added to an over- (resp. under-) approximate summary. For example, replacing $U' = U[p \mapsto \lambda\vec{x}.U[p](\vec{x}) \vee \exists locals.U(body(n))]$ with $U' = U[p \mapsto \lambda\vec{x}.U[p](\vec{x}) \vee \mathrm{QE}(\exists locals.U(body(n))]$) in the UU rule. This use of QE leads to quantifier-free summaries.

When updating over- (resp. under-) approximate summaries, the algorithm can use over- (resp. under-) approximate QE techniques. By comparison, under- (resp. over-) approximate QE would lead to unsoundness. Approximating QE is not only cheaper but can also further simplify the resulting summary.

Selective Updates A final way of preventing summaries from growing too quickly syntactically is by only performing semantic updates. For example, consider O from the goal of a rule that has an update O' in its subgoal. If $\forall\vec{x}.O[p](\vec{x}) \Rightarrow O'[p](\vec{x})$, then although $O'[p]$ contains more conjuncts than $O[p]$, it does not provide any new information. In this case, the algorithm avoids the update and simply uses O in the subgoal instead of O'. Similarly, consider U from the goal of a rule and U' from its subgoal. The algorithm should only update the under-approximation if $\exists\vec{x}.U'[p](\vec{x}) \not\Rightarrow U[p](\vec{x})$. Over-approximate summaries become monotonically more constrained, so if $\forall\vec{x}.O[p](\vec{x}) \Rightarrow O'[p](\vec{x})$ then $\forall\vec{x}.O[p](\vec{x}) \Leftrightarrow O'[p](\vec{x})$. Under-approximations similarly become monotonically less constrained.

4.8 Evaluation and Results

I implemented the algorithm in a tool called CLOVER on top of a CHC solver called FREQHORN [72] and the SMT solver Z3 [61]. I evaluated CLOVER and compared it with existing CHC-based tools on three sets of benchmarks (described later) that comprise standard collections and some new examples that include mutual recursion. I aimed to answer the following questions in the evaluation:

- Is CLOVER able to solve standard benchmarks?
- Is CLOVER more effective than other tools at handling mutual recursion?
- To what extent do EC lemmas help CLOVER solve benchmarks?
- How does the bound k for environments affect CLOVER's performance?

We compared CLOVER against tools entered in the annual CHC-solver competition (CHC-Comp) in 2019: SPACER [103], based on PDR [36]; ELDARICA [95], based on CEGAR [47]; HOICE [39], based on ICE [78]; PCSAT [134]; and ULTIMATE UNIHORN [63], based on trace abstraction [90].

All experiments used a timeout of 10 minutes (as used in CHC-Comp). CLOVER was run on a MacBook Pro, with a 2.7GHz Intel Core i5 processor and 8GB RAM, but the other tools were run using StarExec [145]. CLOVER was not run on StarExec due to difficulties with setting up the tool with StarExec. Note that the platform that CLOVER was run on is less performant than the one on which other tools were run.

4.8.1 Description of Benchmarks

Three sets of varied benchmarks were gathered to evaluate CLOVER. The first set's benchmarks range from roughly 10-7200 lines of SMT-LIB [22], and the latter two sets have smaller but nontrivial code (around 100 lines). The latter two sets were manually encoded into CHCs. Additional details follow.

CHC-Comp. I selected 101 benchmarks from CHC-Comp [40] that were contributed by HoIce and PCSat, since their encodings preserve procedure calls and feature nonlinear CHCs (which can represent procedures with multiple callees per control-flow path)[3].

Real-World. Two families of benchmarks are based on real-world code whose correctness has security implications. The *Montgomery* benchmarks involve properties about the encodings (i.e., Montgomery representations) of numbers used in performing Montgomery multiplication [101]. Each benchmark has an assertion about the sum of Montgomery representations [101] of concrete numbers. The parameters of the Montgomery encoding (i.e., the modulus and auxiliary modulus) are also concrete but vary across benchmarks, avoiding the need for nonlinear arithmetic invariants. The *s2n* benchmarks are based on Amazon Web Services' s2n library [11]. The specifications comprise correct API usage to prevent leaks of private data and involve arrays of unbounded length (not handled by the tool PCSat).

Mutual Recursion. This set of benchmarks containing mutual recursion was created because few CHC-Comp benchmarks exhibit mutual recursion, likely due to lack of tool support. *Even-Odd* benchmarks involve various properties of `e` and `o` (defined as in Sect. 4.1), which is based on examples demonstrating mutual recursion in functional programming languages such as OCaml [119]. For twelve of these benchmarks, the encoding is the same as before, but the properties vary. Twelve additional benchmarks extend `even` and `odd` to handle negative inputs (which lead to more control-flow paths through the procedures).

[3]There is no comparison against FreqHorn since it cannot handle nonlinear CHCs.

Table 4.2: Number of Benchmarks Solved by Clover and Competing Tools.

	Clover				Spacer	Eldarica	HoIce	PCSat	Ultimate Automizer
	k=2	k=9	k=10	k=10, no EC lemmas					
CHC-Comp (101)	80	77	77	72	93	94	92	81	76
Montgomery (12)	0	11	12	12	5	12	12	3	11
s2n (4)	3	4	4	4	3	0	2	N/A	4
Even-Odd (24)	24	24	24	0	12	0	9	0	0
Hofstadter (5)	4	5	4	5	1	4	5	5	0
Mod n (15)	0	15	15	0	0	0	0	0	0
Combination (2)	0	2	2	0	0	0	0	0	0
Total Solved (163)	145	171	171	127	133	110	120	89	91

Another benchmark family is based on the *Hofstadter* Figure-Figure sequence [94]. *Mod* n benchmarks consider mutually-recursive encodings of $\lambda x.x \bmod n = 0$ for $n = 3, 4, 5$, where there are five benchmarks for each value of n. These benchmarks are of particular interest because unlike existing benchmarks, they exhibit greater depths of mutual recursion, as might be found in mutual recursive programs in practice. Experiments were not conducted on such practical programs as recursive descent parsers, since existing front-ends [85, 58] cannot handle them either because they lack of support for recursion [85] or do not support arrays or strings [58]. Because these Mod n benchmarks are parametric, they also serve as a good family of benchmarks for comparison.

Combination benchmarks result from combining Montgomery and Even-Odd benchmarks, where Montgomery representations are used as inputs to `even` and `odd`.

All the mutual recursion benchmarks (except for one of the Hofstadter examples) involve two queries, where the summaries inferred from the first query should (and in Clover's case, do) help with solving the second query.

4.8.2 Results and Discussion

Table 4.2 gives a summary of results. It reports the number of benchmarks solved for each benchmark set by Clover with bound parameter k being 2, 9, and 10 (the best-performing bounds for the three benchmark sets) and by the other tools. It

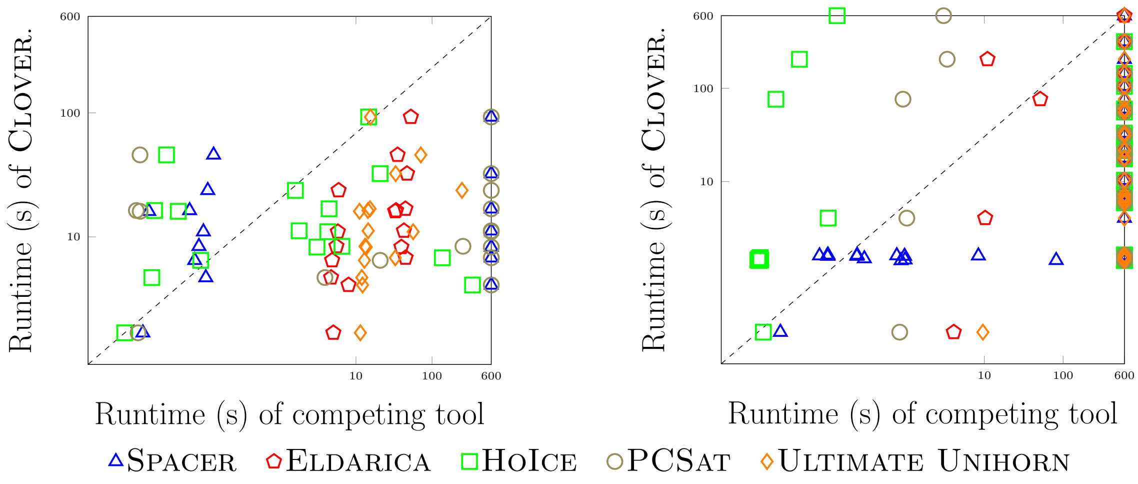


Figure 4.8: Timing results for the Real World (left) and Mutual Recursion (right) benchmarks. Points below the diagonal line are those for which CLOVER outperforms the corresponding tool. Points on the right edge indicate timeouts of the other tool.

also reports results for CLOVER with $k = 10$ but without EC lemmas. Figure 4.8 shows the timing results for other tools against CLOVER for Real-World and Mutual Recursion benchmarks.

4.8.2.1 Efficacy on standard benchmarks.

As can be seen in Table 4.2, CLOVER performs comparably with some other tools on the CHC-Comp benchmarks. It is expected that the performance of CLOVER can be further improved by additional optimizations and heuristics, such as those that improve the quality of interpolants.

4.8.2.2 Efficacy on Mutual Recursion benchmarks.

Table 4.2 and Figure 4.8 demonstrate that CLOVER is more effective and often more efficient at solving Mutual Recursion benchmarks than the other tools. Few tools are able to handle the Even-Odd benchmarks, which CLOVER (with EC lemmas) can solve at any bound value greater than 2. Other tools are unable to solve even half of the Mutual Recursion benchmarks, reinforcing that CLOVER is a useful addition to existing tools that enables handling of mutual recursion as a first class concern.

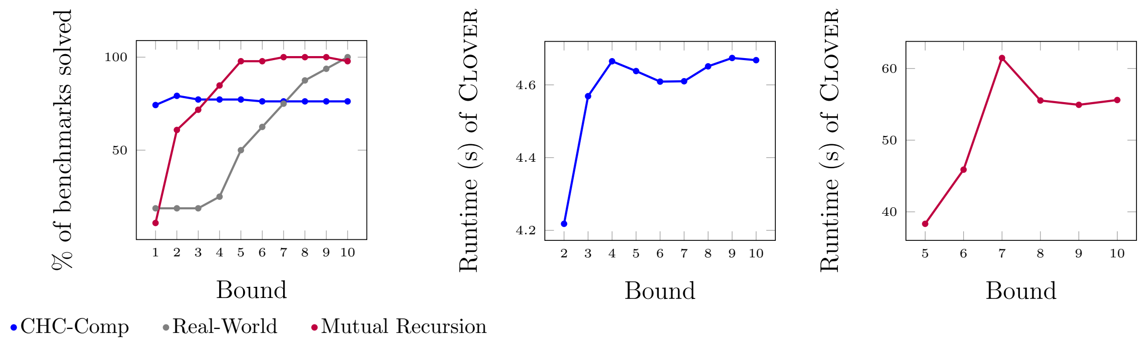


Figure 4.9: Left: Percentage of benchmarks Clover solves with different bounds on different benchmark categories; **Center**, **Right**: Timing results on a representative benchmark from CHC-Comp and Mutual Recursion, respectively.

4.8.2.3 Usefulness of EC lemmas.

Running Clover with and without EC lemmas using bound $k = 10$ revealed their usefulness for many of the benchmarks. In particular, the columns for bound 10 with and without EC lemmas in Table 4.2 show that EC lemmas are needed to allow Clover to solve several CHC-Comp benchmarks and all the Mutual Recursion benchmarks except the Hofstadter ones. These results indicate that Clover's ability to outperform other tools on these benchmarks relies on EC lemmas.

4.8.2.4 Comparison of Different Bounds.

Figure 4.9 (left) shows the number of benchmarks successfully solved by Clover in each set as the bound value is varied. Running Clover with *too small* a bound impedes its ability to prove the property or find a counterexample, since the environment is unable to capture sufficient information. On the other hand, running Clover with *too large* a bound affects the runtime negatively. This effect can be observed in Figure 4.9 center and right, which show how the runtime varies with the bound for a representative benchmark from the CHC-Comp and Mutual Recursion sets, respectively. Note that at a bound $k < 2$, Clover does not solve the given CHC-Comp benchmark, and at $k < 5$, Clover does not solve the given Mutual Recursion benchmark. The best performance for these examples is achieved at the lowest bound size at which Clover can solve the benchmarks, which is as expected, since at these bound sizes, the bounded environments, and thereby the size of the SMT queries, are smaller. As the bound increases, a point is eventually reached where many of the bounded environments that Clover considers are the same as unbounded environments, resulting in less of a performance penalty with increasing bound values. In some cases, the performance may even improve slightly as shown in the rightmost example in Figure 4.9 once all bounded environments become the same as unbounded ones, since the *interface* component of the bounded context becomes trivial to compute. These results confirm the expected trade-off between scalability and environment relevance. The appropriate trade-off – i.e., the best bound parameter to use – depends on the type of program and property. As seen in Figure 4.9 (left), the bound values that lead to the most benchmarks being solved differ per benchmark set. Rather than having a fixed bound, or no bound at all, the ability to choose the bound parameter in Clover allows the best trade-off for a particular set of programs. If the best bound is not known a priori, bound parameters of increasing size can be determined empirically on representative programs.

There is also data on how the number and solving time for each type of SMT query varies with the bound k, averaged over benchmarks in each set. Figure 4.10 shows the statistics on the average number of queries of each type, and Figure 4.11 shows data on the average time taken to solve the query. Here, for the sake of space, OO and UO are used to denote cases where OOIL and UOIL rules are applied. These data are from all runs for which Clover is successful and gives an answer of *safe* or *unsafe*.

We can use these data along with the data in Figure 4.9 to (roughly) compare an approach restricted to $k = 1$ with an approach that allows $k > 1$ in bounded environments. Note that Clover differs significantly in other respects from tools

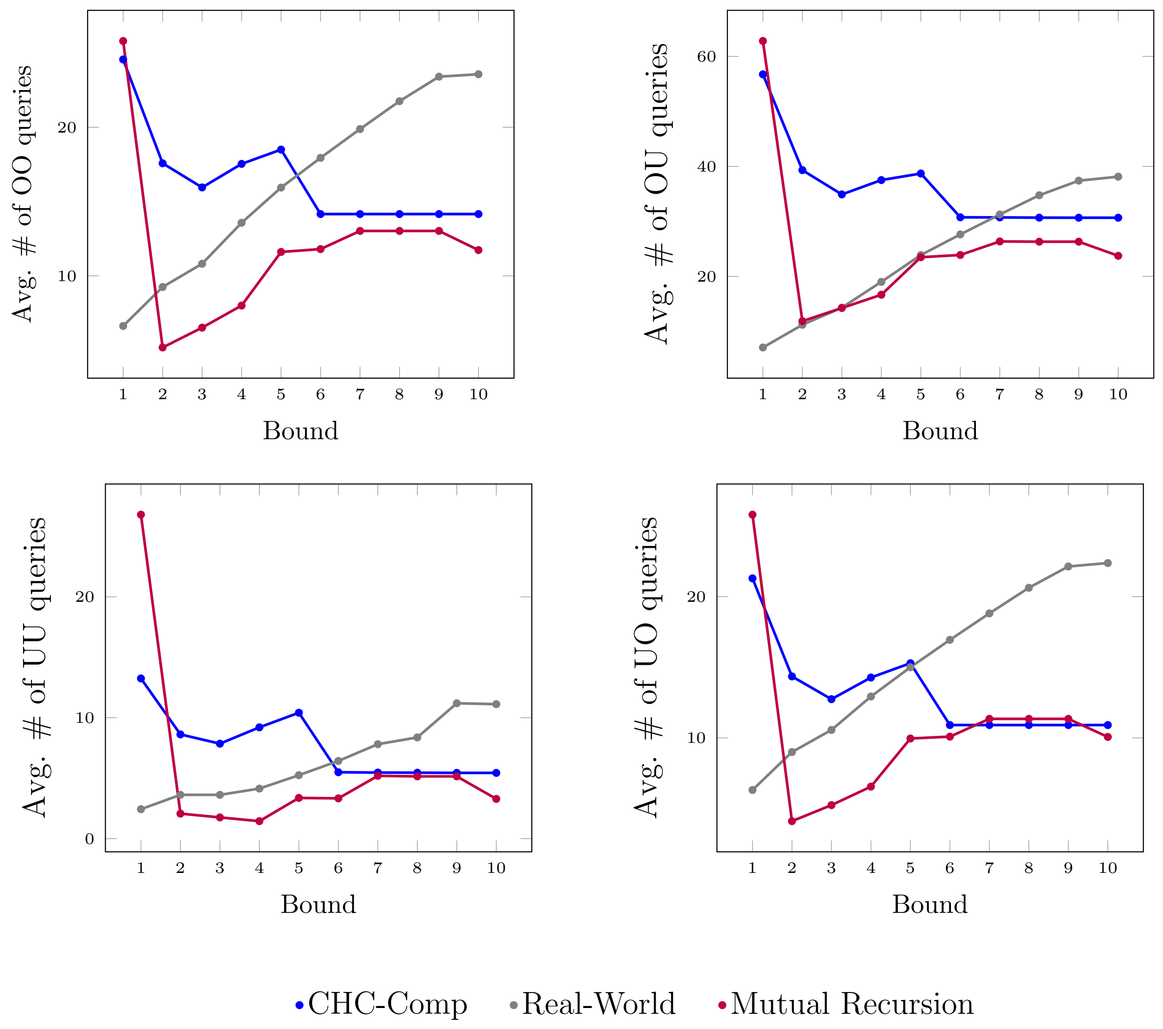


Figure 4.10: Average number of SMT queries made by Clover as the bound changes (for successful runs).

like Spacer and Smash that enforce $k = 1$ in environments[4], making it difficult to perform controlled experiments to compare this aspect alone.

Note from Figure 4.10 that for the CHC-Comp and Mutual Recursion sets of benchmarks, the number of SMT queries of all types is lower at $k > 1$ in comparison to $k = 1$. This result indicates that benchmarks that can be solved with $k > 1$ require on average fewer updates to procedure summaries than are needed on average for benchmarks that can be solved with $k = 1$, confirming the benefit of improved relevance when going beyond a restricted environment with $k = 1$. The data for the

[4]Unlike Spacer it does not use PDR to derive invariants, and unlike Smash it is not limited to predicate abstractions.

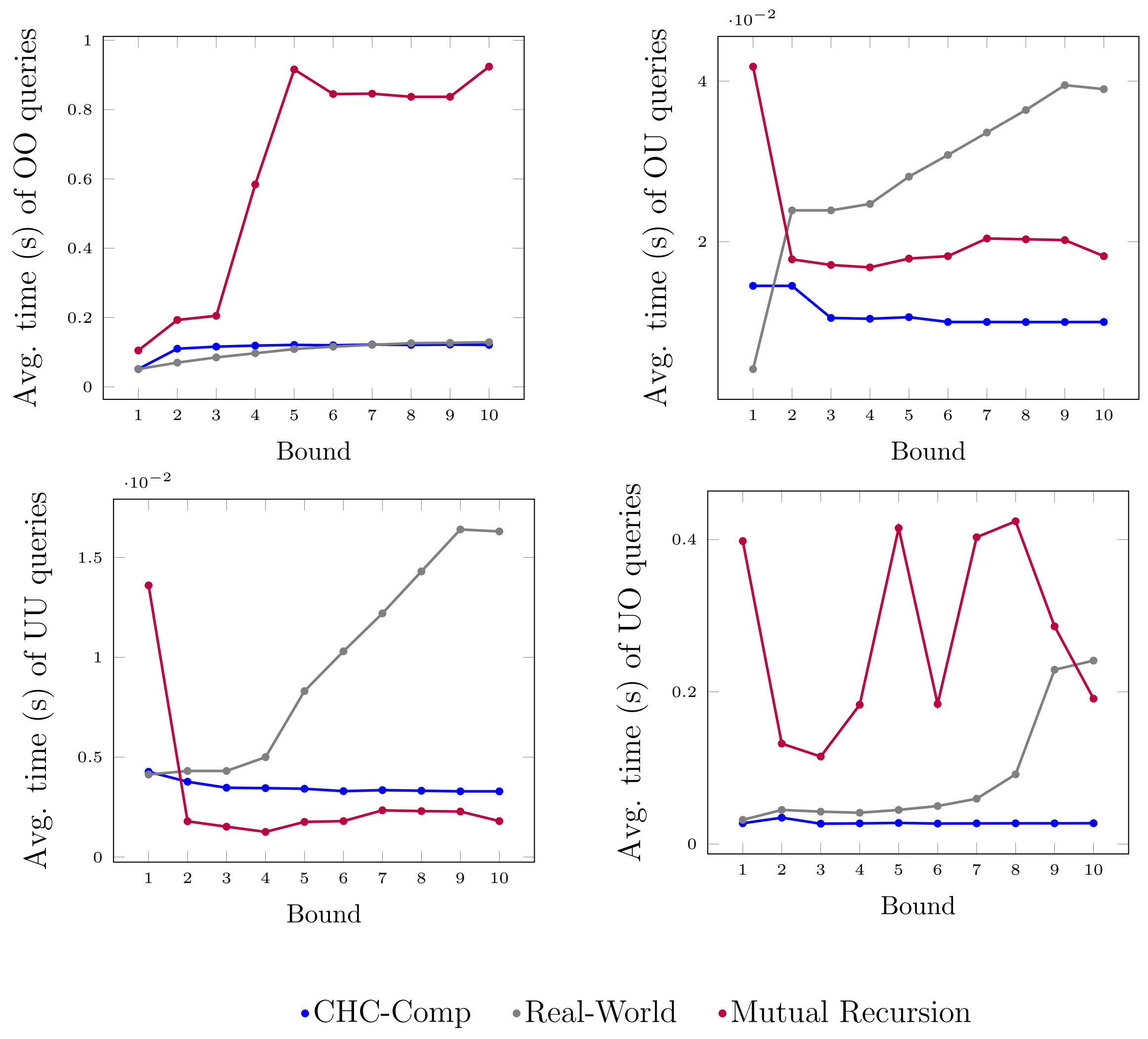


Figure 4.11: Average solve times of SMT queries made by Clover as the bound changes (for successful runs).

Real-World does not follow this trend because a higher bound ($k = 10$) is needed to solve the examples (as can be seen in Figure 4.9).

From Figure 4.11, it is clear that the OU and UU queries are cheaper than OO and UO queries, which is expected since the latter require over-approximating the target's body. Unsurprisingly, OO queries are the most expensive. Average times of non-OO queries for $k > 1$ are lower than (or about the same as) average times for $k = 1$ for the CHC-Comp and Mutual Recursion sets but continue to increase with k in the Real-World set because solving the Montgomery benchmarks relies on propagating under-approximations from increasingly large call graph depths.

4.9 Related Work

There is a large body of existing work that is related to this chapter in terms of CHC solving, program analysis, and specification inference.

4.9.1 CHC Solving

As mentioned in Chapter 2, program verification problems, including modular verification, can be encoded into systems of CHCs [84, 112, 85, 58]. Since the encoding of program verification problems into systems of CHCs was proposed, the approach has received a lot of attention within the research community, yielding many existing tools for solving systems of CHCs [39, 103, 111, 149, 73, 43, 84, 95, 157].

A few techniques can be seen as using bounded or unbounded environments. In particular, such notions are directly applicable to other deductive techniques for CHC solving that use unfolding.

For example,SPACER [103], which is based on PDR [36, 65], can be seen as considering bounded environments but only allowing a bound of one ($k = 1$). Meanwhile, DUALITY [112] can be seen as considering only unbounded environments. The difference between DUALITY and a PDR-like approach with respect to how much of the environment is used has been referred to as the *variable elimination trade-off* [111], where eliminating too many variables can lead to over-specialization of learned facts (PDR) and eliminating no variables can lead to larger subgoals (DUALITY). The ability to enable a trade-off between these two extremes served as motivation for the development of the notion of parameterizable bounded environments introduced in this chapter.

Algorithms in HSF [84], ELDARICA [95] and LIQUIDHASKELL [150] use counterexample-guided abstraction refinement (CEGAR). Initial summaries are inferred by using predicate abstraction and then refined by adding new predicates derived from potentially-spurious counterexamples. Such counterexamples are analogous to unbounded environments, and these tools may face scalability issues depending on how environments and procedures are considered. For example, ELDARICA makes disjunctive interpolation queries that grow larger with the number of procedures involved in counterexamples [95].

Other techniques do not have any notions that correspond even analogously to bounded environments. HOICE [39], FREQHORN [73], and LINEARARBITRARY [157] are based on guessing summaries using machine learning and/or SyGuS techniques [8]. All of these approaches have trade-offs between scalability of the search space of guesses and expressivity of guessed summaries.

CLOVER has many algorithmic differences from these efforts. Most existing tools do not place any bounds on the environments (if they are used at all). This includes approaches that unfold a relation at each step [111, 149] and CEGAR-based approaches [84, 95] where counterexamples can be viewed as environments. These tools face scalability issues as environments grow; DUALITY makes larger interpolation queries as more relations are unfolded [112], and ELDARICA makes larger tree/disjunctive interpolation queries for counterexamples that involve more procedures [95]. CLOVER also has significant differences with SPACER, which, as mentioned previously, can be seen as using a bounded environment of $k = 1$, Other than CLOVER having the ability to adjust the bound for the environment, another significant difference between SPACER and CLOVER is that the former uses PDR-style bounded assertion maps to perform induction, whereas CLOVER uses induction explicitly and derives EC lemmas.

Most of the mentioned tools also do not learn implications that relate procedure summaries. DUALITY may implicitly use assumptions, and a couple of other tools [39, 149] do learn lemmas with implications, but none of them learn lemmas that, like EC lemmas, are in forms that are well-suited for handling mutual recursion. The use of

Syntax-Guided Synthesis [8] to search for procedure summaries is orthogonal to the verification algorithm implemented in Clover; both approaches can be effectively combined, as will be shown in Chapter 5, where the tool Flower uses SyGuS-based techniques with and within the Clover algorithm to verify information-flow security properties.

4.9.2 Program Analysis and Verification

As mentioned, encoding a program into a systems of CHCs and applying a CHC solver provides one avenue for implementing a program verifier, with several such program verifiers including a front-end for converting a program into a system of CHCs and using any one of the CHC-solving methods described above in the backend to solve the resulting system [84, 112, 85, 58]. Other than CHC-based methods, there are techniques such as abstract interpretation [51, 52, 69] and interprocedural dataflow analysis [136, 128] that can infer procedure summaries and perform modular verification. These approaches often use fixed abstractions and path-insensitive reasoning, which may result in over-approximations that are too coarse for verification.

The software model checker Bebop [18] in SLAM [19] extended interprocedural dataflow analysis with path sensitivity. It was used successfully to verify device drivers in SLAM [19], a pioneering work in software model checking that inspired new generations of verifiers, including the CHC solvers and several other program verifiers mentioned in this chapter. Related model checkers include a direct precursor to Duality [112, 110] and other adaptations of PDR to software [93, 45], which can be seen as precursors to Spacer [103]. Of these, GPDR [93] is similar to Spacer, but lacks modular reasoning and under-approximations. Specification inference (including Houdini-style learning [77]) has also been used to enable modular verification of relational programs [104] (and is employed in Flower as well, as described in the next chapter).

Another relevant tool is the modular verifier Smash [82]. It uses context-sensitive over- and under-approximate procedure summaries, and alternation between them. Smash demonstrated the benefits of maintaining over- and under-approximate summaries that interact with one another, inspiring later work such as that on Spacer [103] and Clover to use both over- and under-approximate summaries and to allow them to influence each other. Smash has an analogous concept to an environment for a procedure call, where the "environment" for a procedure call is expressed as a pair of a precondition and a postcondition, where the former is an under-approximation of the program execution preceding the call, and the latter is an over-approximation of the program execution following the call. These environments can be viewed as analogous to bounded environments with a fixed bound of 1, similar to Spacer's. Note that in contrast to summaries found by CHC-solving methods, which are often arbitrary first-order logic interpretations of uninterpreted predicates representing the procedure calls, procedure summaries in Smash are of the form of a pair of a precondition and postcondition, each of which is comprised of predicate abstractions. In contrast, CHC solvers can provide summaries that are richer formulas in first-order logic theories and need not rely on predicate abstraction unlike Smash and other related tools [82, 81, 86].

4.9.3 Specification Inference

Many past efforts [9, 12, 30, 126, 140, 153] focused on learning coarse interface specifications or rules specifying the correct usage of library API calls, rather than learning logical approximations of procedures. Other specification inference techniques learn procedure summaries for library API procedures by using abstract interpretation [91, 52] or learn information-flow properties about the procedures [108].

Other related work [4] infers maximal specifications for procedures with unavailable bodies, and other techniques assume an angelic environment setting [35, 57]

– specifications inferred by these techniques may not be valid over-approximations. The recent HornSpec [131] performs inference of maximal non-vacuous specifications for inputs provided as systems of CHCs; it relies on forward and backward reasoning in the style of deductive CHC-solving techniques like the one presented in this chapter and Spacer [103]. Another technique [15] also uses interpolation to infer over-approximate summaries and leverages different SMT theories for scalability but is not applicable to recursive programs.

Chapter 5

Verification of information flow security

The problem of verifying secure information flow is that of guaranteeing that a program does not leak private inputs to public outputs. To solve this problem, one can verify non-interference [83]: for any two runs of a program with the same public inputs but possibly different private inputs, the public outputs of the program are equal. As mentioned in Chapter 2, this property is an instance of a *2-safety hyperproperty*, i.e., a relational property involving more than one execution of the same program. In practice, non-interference is often too strong a property to enforce. For example, a password recognizer would have its public output be influenced by whether or not the user-provided private input is the correct password. A common approach is to allow values that need to be leaked to be *declassified* [132].

Barthe *et al.* proposed verifying secure information flow by reducing it to safety verification on a product or self-composed program [28]. Despite advancements in automated program verifiers, the ability to perform successful safety verification in practice can depend critically on how the product program is constructed. Construction of product programs has thus been a focus in subsequent efforts [148, 27, 25, 26, 89, 48, 59, 41, 42, 155, 139], as explained in more detail Chapter 2. These efforts encompass various syntactic and semantic transformations, heuristics, and use of reinforcement learning for constructing suitable product programs. Some relational property verifiers avoid explicitly constructing product programs altogether [143, 32, 29, 31, 70, 13].

This chapter addresses a related but distinct limitation of existing efforts based on reduction to safety. Most such techniques are non-modular, i.e., they neither leverage nor infer *relational specifications for procedures* in interprocedural programs. In general, modular verification offers significant benefits over non-modular techniques – it is inherently more scalable, can provide procedure interface contracts (not only verification results), and can improve code understanding and maintenance. For example, relational specifications of procedures can provide security contracts for library APIs, such as in the s2n implementation of the TLS protocol [11].

A few other approaches do leverage relational specifications of procedures, but they either restrict both copies of the program to always follow the same control flow [25] or are not automated [66, 89]. In particular, the work by Eilers *et al.* [66] proposes a *modular product program (MPP)* construction, which is suitable for performing *modular* relational program verification. Intuitively, this enables reduction to safety on a per-procedure basis without constructing a monolithic product program. In their implementation, the Viper language was extended to support information flow specifications, and Viper back-end verifiers checked secure-information-flow properties on benchmarks, but each procedure required user-provided relational invariants and related annotations rather than relying on tools to derive them automatically. Placing this annotation burden on users becomes a barrier to automated verification.

As seen in the previous chapter, deriving sufficient relational invariants for procedures is a challenging problem, and existing off-the-shelf safety verifiers [77, 5, 130, 103, 121, 72, 157] may not be able to infer them. For verifying secure information flow, such invariants often have a special form that is unlikely to be produced by standard interpolation and existing heuristics in these verifiers. For example, experimental results (§5.7) show that Spacer often fails to infer invariants needed to verify information flow in programs with recursion. The approach by Eilers et al. requires users to provide such invariants and other related annotations, e.g., which procedure inputs and outputs are private or public. The resulting verification conditions can then be checked by theorem-provers such as SMT solvers.

In this chapter, I propose to augment the approach to modular program verification described in Chapter 4 with Syntax-Guided Synthesis (SyGuS) [8] to automatically infer useful relational specifications about information flow in procedures of a MPP. The structure in information-flow specifications makes them suitable targets for grammar-based enumerative search and synthesis. These inferred specifications not only reduce the aforementioned annotation burden for proving secure information flow but also aid in code understanding. The choice to work with MPPs is motivated by two aspects: they enable *modular* relational verification and they allow leveraging existing techniques for construction of suitable product programs within each procedure. An MPP can be encoded into a set of CHCs as any other program with safety annotations, and the approach outlined here, when run on these CHCs, automatically infers relational specifications that are sufficient for verifying the program with respect to given security properties. If there are no given security properties, our approach can still infer relational specifications for procedures that are useful for code understanding or subsequent verification.

My SyGuS-based approach is based on an enumerative search using grammars over a combination of tagged variables, program syntax (as preserved in the CHC encoding), and certain kinds property templates derived from the syntax of a predicate application's environment. Enumerate-and-check approaches have been shown to be effective for synthesis of quantifier-free invariants [120, 10, 72, 129] and more recently quantified invariants for CHCs handling arrays [74]. This work shows that such an approach is also effective for information-flow properties.

This work proposes three templates to generate grammars for invariant synthesis: one that expresses quantifier-free information-flow properties, and two that express quantified properties, which are often difficult to handle by existing automated verifiers. Of the latter two, one infers quantified information-flow properties over *arrays*, and the other infers specifications involving the *context* in which a procedure is called (as represented by the corresponding predicate application's environment), making this template well-suited for inferring properties where declassification has occurred prior to the procedure being called, since the declassified values will be low-security in the callee.

I have implemented this approach in a tool called FLOWER. An evaluation on available benchmark examples demonstrates that it is effective in inferring useful relational specifications of procedures, without requiring any user-provided annotations. I have also compared FLOWER with other state-of-the-art tools: the hyperproperty verifier DESCARTES [143] and the modular CHC-based verifier SPACER [103]. Experiments demonstrate that FLOWER generally outperforms them, especially on benchmark examples that contain loops or recursion.

The work presented in this chapter has previously been presented and published at a conference [124].

5.1 Motivating Example

This section demonstrates this approach on an example program `P` shown in Figure 5.1, inspired by a related work [50]. Though the technique itself operates on a CHC encoding of the program, the description in this section will be in terms of the program before the encoding takes place in order to provide intuition.

```
main (int[] a, int n) {
  a := init(a, 0);
  outputter(a, 0);
  return n;
}


init(a, i) {
  if (i ≥ 64) return a;
  declassify(a[i] = 0);
  return init(a, i + 1);
}


outputter(a, i) {
  if (i ≥ 64) return;
  if (a[i] = 0) {
    assert(low(a[i]));
    print(a[i]);
  }
  outputter(a, i + 1);
}
```

```
main (b1, b2, a1, a2, n1, n2) {
  a1, a2 := init(b1, b2, a1,
                 a2, 0, 0);
  assert(outputter(b1, b2, a1,
                   a2, 0, 0));
  return n1, n2;
}


init(b1, b2, a1, a2, i1, i2) {
  if (¬(b1 ∧ i1 < 64 ∨
             b2 ∧ i2 < 64))
    return a1, a2;
  l1 := b1 ∧ i1 < 64
  l2 := b2 ∧ i2 < 64
  assume (l1 ∧ l2 ⇒
    (a1[i1] = 0) = (a2[i2] = 0));
  return
    init(l1, l2, a1,
         a2, i1 + 1, i2 + 1);
}


outputter(b1, b2, a1, a2, i1, i2) {
  if (¬(b1 ∧ i1 < 64 ∨
        b2 ∧ i2 < 64))
    return true;
  l1 := b1 ∧ i1 < 64;
  l2 := b2 ∧ i2 < 64;
  t1 := l1 ∧ a1[i1] = 0;
  t2 := l2 ∧ a2[i2] = 0;
  print(t1, t2, a1[i1], a2[i2]);
  ok := t1 ∧ t2 ⇒ a1[i1] = a2[i2];
  ok := ok ∧
        outputter(b1, b2, a1, a2,
                  i1 + 1, i2 + 1);
  return ok;
}
```

Figure 5.1: Example (**left**: original (`P`), **right**: modular product program (`MP`)).

In `main`, a call to `init` makes initial assumptions about the array `a`: for each of the first 64 values in the array, the information about whether or not the value is 0 is declassified recursively. Then, these 64 entries are printed out by the recursive procedure `outputter`, which contains an assertion that checks that each of the values printed out is public (i.e., low-security) output. Finally, `main` returns its second argument.

The security primitives used in this example are `low`, which is a predicate that holds iff its argument is a low-security variable, and `declassify`, which has the effect of making the value of its argument low-security after the point where `declassify` is invoked. Without assumptions stating otherwise (i.e., either `assume` statements that indicate that a value is low-security by using the `low` primitive or `declassify` statements), we assume that all inputs are high-security. In the example, after `init` is called and it declassifies each `a[i] = 0` value for `i < 64`, then the information about whether or not any of the first 64 entries in `a` is 0 is considered to be public information. The `outputter` procedure prints out the value of values of `a[i]` for `i < 64` only under the condition that `a[i] = 0`. This behavior leaks exactly only the declassified information, so the assertion is expected to hold for each call to `outputter`.

The modular product program `MP` for this example is shown in Figure 5.1 (right). Note that for *each* variable in `P` (even if irrelevant to verification), `MP` has two copies reflecting the two executions of the program, e.g., `n` is translated to `n1` and `n2`. For each procedure in `P`, two Boolean *activation variables* `b1` and `b2` are added as inputs to the corresponding procedure in `MP`, where they respectively indicate whether the control flow in the corresponding copy of the program has reached the callsite. The idea is that *relational specifications* for procedures hold when both copies of the program have reached the same callsite, i.e., when both activation variables of the callee are true. As a result, all the relational specifications that we infer are implications in which the antecedent contains at least `b1` and `b2` as conjuncts.

The translation to `MP` also shows how the information-flow operation `declassify` is encoded as an assumption, and how the information-flow specification `low(a[i])` is translated into a relational property $\mathtt{t1} \wedge \mathtt{t2} \Rightarrow \mathtt{a1}[\mathtt{i1}] = \mathtt{a2}[\mathtt{i2}]$ in `MP`, where `t1` and `t2` were the activation variables under which the specification `low(a[i])` occurred.

Finally, note that the assertion in `outputter` has been hoisted [105] to `main` in `MP`, with the return value of `outputter` being `true` if and only if no assertion failed.

The proposed technique will infer quantifier-free information-flow properties for each procedure (i.e., procedure summaries that involve information-flow information). For example, the technique can infer that for `main` of `MP`, the following property holds, where res_1 and res_2 represent the return values of `main`:

$$\mathtt{b1} \wedge \mathtt{b2} \wedge \mathtt{n1} = \mathtt{n2} \Rightarrow \mathit{res}_1 = \mathit{res}_2$$

This property says that the output of `main` depends only on its second argument, and it does not rely on any information about whether the second argument or output of `main` is public or private, nor does it express any such information.

The technique also infers quantified invariants, e.g., $\phi(\mathtt{i1})$:

$$\forall j_1, j_2 . \mathtt{i1} \leq j_1 \leq 64 \wedge j_1 = j_2 \Rightarrow (\mathtt{a1}[j_1] = 0) = (\mathtt{a2}[j_2] = 0)$$

It can then instantiate this property for the call to `init` in `main` to determine that $\phi(0)$ is true when the call to `outputter` is made. However, it cannot yet verify the program because at this point we have not inferred sufficient properties for `outputter`.

Finally, it uses the context in which `outputter` is called to influence the guesses that we make for the antecedent in its relational specification. Then it infers the following property for `outputter`, where *res* is the return value of `outputter`:

$$\mathtt{b1} \wedge \mathtt{b2} \wedge \phi(0) \wedge \mathtt{i1} = \mathtt{i2} \wedge 0 \leq \mathtt{i1} \Rightarrow \mathit{res}$$

Note that this property contains quantifiers because $\phi(0)$ does. This property enables us to verify that the assertion for the program holds, leading to a successful conclusion.

$$
\begin{array}{ll}
\textit{proc} & ::= P(x_1, \ldots, x_m)\ \texttt{returns}\ (y_1, \ldots, y_n)\ \{\textit{stmt}\} \\
\textit{stmt} & ::= \textit{stmt}_1; \textit{stmt}_2 \mid (x_1, \ldots, x_n) := e \mid \texttt{assert}\ e \mid \texttt{assume}\ e \mid \texttt{havoc}\ x \mid \\
 & \quad \texttt{if}(e)\ \{\textit{stmt}\} \mid (x_1, \ldots, x_n) := P(e_1, \ldots, e_m)
\end{array}
$$

Figure 5.2: The syntax of procedures being considered for the MPP transformation

5.2 Modular Product Programs

This section will provide a brief description of product programs to make more explicit the intuition gained by examining the MPP in Figure 5.1. Recall that a k-hyperproperty expresses a property over k runs of the same program. Product programs convert k-hyperproperties into safety properties by creating k renamed versions of all the original variables. In contrast to ordinary product programs, *modular product programs (MPP)* avoid duplicating control structures such as procedure calls by introducing Boolean *activation variables* that indicate whether each program copy has reached a certain execution point [66]. The current activation variable for copy i is true if and only if copy i is currently at that location. Since this chapter is concerned only with 2-hyperproperties, the remainder of this explanation will be specifically for the construction of MPPs for 2-hyperproperties. This construction generalizes naturally to $k > 2$, and the full definition of how to construct a MPP for arbitrary k can be found in the original work on MPPs [66].

For a modular product program with k copies, partial functions *idx* and *getIdx* conveniently handle expressions with renamed copies of variables. For any expression e, $\textit{getIdx}(e) = i$ iff e represents an expression only over variables from the i^{th} copy; and for any expression e such that $\textit{getIdx}(e)$ is defined: $\textit{getIdx}(\textit{idx}(e, i)) = i$. For example, $\textit{idx}(\texttt{b} \wedge \texttt{i} < 64, 2) = \textit{idx}(\texttt{b1} \wedge \texttt{i1} < 64, 2) = \texttt{b2} \wedge \texttt{i2} < 64$. We also use *idx* to denote the lifting of *idx* to sets of expressions.

Let Figure 5.2 denote the syntax of a procedure P. Program variables are denoted by x and y and expressions by e. Note that loops are not included here because they can be encoded as recursive procedures (and eventually will need to be for the later CHC encoding). Similarly, conditionals with `else` branches are not included, since an else branch can be captured with another `if` statement. Figure 5.3 gives the transformation PP that converts a procedure *proc* into a MPP procedure with the same name *proc*. Note that $PP^{\mathtt{b}}$ transforms program statements to suitable ones for a MPP using $\mathtt{b}_1$ and $\mathtt{b}_2$ as activation variables. Fresh activation variables are introduced for every procedure and for any scopes within branches. Any procedure call uses the innermost scope's activation variables. All variables introduced by PP are assumed to be fresh variables (e.g., $\mathtt{b}'_1$, $\mathtt{b}'_2$, $a_1, \ldots, a_m$, $t_1, \ldots, t_n$), and $alt(x_1, \ldots, x_n)$ is short for $idx(x_1, 1), idx(x_1, 2), \ldots, idx(x_n, 1), idx(x_n, 2)$.

The transformation shown in Figure 5.1 has additional modifications than the application of PP to each procedure. First, the assertion is handled differently than described in Figure 5.3 because it is an *information-flow-related* assertion. Such assertions are left alone and expanded into relational assertions after the MPP transformation. Similarly, there is a `declassify` statement in the program that is expanded into relational assumption; the details of how this is performed are provided in the subsequent section, but this expansion occurs after the MPP transformation. Finally, during the encoding of a program into a set of CHCs, *assertion hoisting* is applied [105]. The MPP shown in Figure 5.3 has had information-flow-related assertions hoisted and a Boolean variable `ok` introduced to keep track of whether or not an assertion violation has occurred.

5.3 Secure Information Flow

We use a standard reduction [28] of a (termination-insensitive) secure-information-flow property to a 2-hyperproperty called *non-interference* [83], which ensures that private inputs do not impact public outputs. For a procedure `f`, this is formalized as

$$
\begin{array}{ll}
PP(P(x_1,\ldots,x_m)\{stmt\}) & = P(b_1,b_2,alt(x_1,\ldots,x_m))\ \texttt{returns}\ alt(y_1,\ldots,y_n) \\
 & \qquad \{PP^{\mathtt{b}}(stmt)\} \\
PP^{\mathtt{b}}(stmt_1;stmt_2) & = PP^{\mathtt{b}}(stmt_1); PP^{\mathtt{b}}(stmt_2) \\
PP^{\mathtt{b}}((x_1,\ldots,x_n):=e) & = \texttt{if}(\mathtt{b}_1)\ \{idx((x_1,\ldots,x_n),1) := idx(e,1)\}; \\
 & \quad \texttt{if}(\mathtt{b}_2)\ \{idx((x_1,\ldots,x_n),2) := idx(e,2)\} \\
PP^{\mathtt{b}}(\texttt{assert}\ e) & = \mathit{if}(\mathtt{b}_1)\ \{\texttt{assert}\ idx(e,1)\}; \\
 & \quad \mathit{if}(\mathtt{b}_2)\ \{\texttt{assert}\ idx(e,2)\} \\
PP^{\mathtt{b}}(\texttt{assume}\ e) & = \mathit{if}(\mathtt{b}_1)\ \{\texttt{assume}\ idx(e,1)\}; \\
 & \quad \mathit{if}(\mathtt{b}_2)\ \{\texttt{assume}\ idx(e,2)\} \\
PP^{\mathtt{b}}(\texttt{havoc}\ x) & = \mathit{if}(\mathtt{b}_1)\ \{\texttt{havoc}\ idx(x,1)\}; \\
 & \quad \mathit{if}(\mathtt{b}_2)\ \{\texttt{havoc}\ idx(x,2)\} \\
PP^{\mathtt{b}}(\texttt{if}(e)\ \{stmt\}) & = \mathtt{b}'_1 := b_1 \wedge idx(e,1); \\
 & \quad \mathtt{b}'_2 := b_2 \wedge idx(e,2); \\
 & \quad PP^{\mathtt{b}'}(stmt) \\
PP^{\mathtt{b}}((x_1,\ldots,x_n) := & \\
\qquad P(e_1,\ldots,e_m)) & = \mathit{if}(\mathtt{b}_2 \vee \mathtt{b}_2)\{ \\
 & \qquad \texttt{if}(\mathtt{b}_1)\{idx((a_1,\ldots,a_m),1) := idx((e_1,\ldots,e_m),1)\}; \\
 & \qquad \texttt{if}(\mathtt{b}_2)\{idx((a_1,\ldots,a_m),2) := idx((e_1,\ldots,e_m),2)\}; \\
 & \qquad (alt(t_1,\ldots,t_n)) := P(\mathtt{b}_1,\mathtt{b}_2,alt(a_1,\ldots,a_m)); \\
 & \qquad \texttt{if}(\mathtt{b}_1)\{idx((x_1,\ldots,x_n),1) := idx((t_1,\ldots,t_n),1)\}; \\
 & \qquad \texttt{if}(\mathtt{b}_2)\{idx((x_1,\ldots,x_n),2) := idx((t_2,\ldots,t_n),2)\}; \\
 & \quad \}
\end{array}
$$

where for any $\vec{y} = y_1,\ldots,y_n$, $alt(\vec{y})$ is the tuple $(idx(y_1,1),\ldots,idx(y_n,1),idx(y_1,2),\ldots,idx(y_n,2))$

Figure 5.3: How to transform a procedure into a procedure in a MPP

follows:

$$\forall \vec{li}_1, \vec{lo}_1, \vec{hi}_1, \vec{ho}_1, \vec{li}_2, \vec{lo}_2, \vec{hi}_2, \vec{ho}_2.$$
$$(\vec{lo}_1, \vec{ho}_1) = \mathtt{f}(\vec{li}_1, \vec{hi}_1) \land (\vec{lo}_2, \vec{ho}_2) = \mathtt{f}(\vec{li}_2, \vec{hi}_2) \land \vec{li}_1 = \vec{li}_2 \Rightarrow \vec{lo}_1 = \vec{lo}_2$$

Variables $\vec{li}_1$ and $\vec{li}_2$ represent public inputs to `f` and $\vec{lo}_1$ and $\vec{lo}_2$ represent public outputs. Variables $\vec{hi}_1$ and $\vec{hi}_2$ represent private input variables to `f` and $\vec{ho}_1$ and $\vec{ho}_2$ represent private outputs. Non-interference states that for any two runs of `f`, one with inputs $\vec{li}_1, \vec{hi}_1$ and one with inputs $\vec{li}_2, \vec{hi}_2$, if their public inputs are equal (i.e., $\vec{li}_1 = \vec{li}_2$), then their public outputs should be equal (i.e., $\vec{lo}_1 = \vec{lo}_2$) regardless of the private inputs' values.

In a modular product program, relational properties become properties over a single run and take the form of an implication whose antecedent implies the truth of all activation variables, e.g., non-interference takes the following shape:

$$\forall b_1, b_2, \vec{li}_1, \vec{lo}_1, \vec{hi}_1, \vec{ho}_1, \vec{li}_2, \vec{lo}_2, \vec{hi}_2, \vec{ho}_2\ .$$
$$b_1 \land b_2 \land (\vec{lo}_1, \vec{ho}_1, \vec{lo}_2, \vec{ho}_2) = \mathtt{f}(b_1, b_2, \vec{li}_1, \vec{li}_2, \vec{hi}_1, \vec{hi}_2) \land \vec{li}_1 = \vec{li}_2 \Rightarrow \vec{lo}_1 = \vec{lo}_2$$

Assertions such as $\mathtt{assert}(\mathtt{low}(e))$ state that the value of expression e should be treated as a public output, and can be expanded in the MPP as $\mathtt{b}_1 \land \mathtt{b}_2 \Rightarrow \mathtt{assert}(\mathit{idx}(e, 1) = \mathit{idx}(e, 2))$, where $\mathtt{b}_1$ and $\mathtt{b}_2$ are the activation variables for the innermost scope containing the `assert` statement.

Requiring non-interference can be restrictive since programs may need some amount of leakage to exhibit the desired behavior. *Declassification* can allow secure-information-flow properties to be checked even for programs that leak some information about high-security variables. Declassification is encoded in modular product programs as an assumption that if both programs reach the same `declassify` statement (i.e., if both activation variables are true), then the value being declassified is equal across both copies of the program. Thus $\mathtt{declassify}(e)$ is encoded as $\mathtt{assume}\ b_1 \land b_2 \Rightarrow e_1 = e_2$ in the MPP. Because private values are encoded as above as the absence of knowledge about whether an expression is equal across executions, this encoding is sound [66].

5.4 SyGuS-based Summary Inference

This section describes the SyGuS-based algorithm for inferring procedure summaries of modular product programs and using these summaries for verification. It takes CHCs as input and maintains a mapping O from uninterpreted predicates in the CHCs to inductive interpretations (recall the notation and definitions for CHCs from Chapter 2). The algorithm updates O as it runs and maintains the invariant that M's interpretations are inductive. The algorithm also contains a mapping U from predicates to under-approximate summaries, but this is only updated within the GetPDGuesses procedure.

The top-level procedure (Algorithm 9) begins with an initial mapping O from each n-ary predicate $R \in \mathcal{R}$ to the coarsest interpretation possible and an initial mapping U from each n-ary predicate $R \in \mathcal{R}$ to the most restrictive interpretation possible. In pseudocode, CheckGuesses(G, O, R) refers to an iterative procedure over all CHCs, where each application $R(\vec{x})$ of symbol R is replaced by formula $\lambda \vec{x}. O[R](\vec{x}) \wedge \mathit{makeGuess}(G)(\vec{x})$, where G is a set of guessed interpretations for R based on grammar templates and $\mathit{makeGuess}(G) = \lambda \vec{x}. \bigwedge\{g(x) \mid g \in G\}$.

The CHCs after the replacement are checked for validity using an SMT solver: if for some CHC C, the corresponding implication does not hold, then the current interpretations for R (which must appear in C) are weakened in G (using, e.g., the Houdini algorithm [77]), and the internal loop in CheckGuesses is repeated. A new inductive mapping O' is returned as the result of CheckGuesses, where $O'[R] = \lambda \vec{x}. O[R](\vec{x}) \wedge \mathit{makeGuess}(G')(\vec{x})$, where G' is the subset of the original set G passed as an argument to CheckGuesses that resulted from any weakening from the internal

Algorithm 9 Top-level verification and summary inference procedure.

```
1: procedure InferSum(CHCs $\mathcal{C}$ with uninterpreted predicates $\mathcal{R}$)
2:     for $R \in \mathcal{R}$ do $O[R] \leftarrow \lambda x_1, \dots, x_n.\top$
3:     for $R \in \mathcal{R}$ do $U[R] \leftarrow \lambda x_1, \dots, x_n.\perp$
4:     for $C \in \mathcal{C}$ where $C = body \Rightarrow R(\vec{x})$ do
5:         $G \leftarrow$ GetQFGuesses($C$) $\cup$ GetQuantifiedGuesses($C$)
6:         $O \leftarrow$ CheckGuesses($G$, $O$, $R$)
7:     while $O$ is not a solution for $\mathcal{C}$ do
8:         $Q \leftarrow$ GetUnsatisifiedQuery($\mathcal{C}$)
9:         $\mathcal{C}' \leftarrow \mathcal{C} \setminus$ GetUnsatisfiedQueries($\mathcal{C}$)
10:        $(result, Goal) \leftarrow$ VerifyWithGuesses($\mathcal{C}'$, $O$, $U$, $Q$)
11:        $O \leftarrow O$ in $Goal$
12:        $U \leftarrow U$ in $Goal$
13:        if $result = unsafe$ then
14:            return $(unsafe, (O, U))$
15:    return $(safe, (O, U))$
```

Algorithm 10 Inference procedure for property-directed guesses.

```
1: procedure RefineOver($M$, $n$)
2:     for disjunct $d$ in $body(n)$ do
3:         $G \leftarrow$ GetPDGuess($M(benv(n))$, $d$, $O$)
4:         $O' \leftarrow$ CheckGuesses($G$, $O$, $R$)
5:         if $O' \neq O$ then return $O'$
6:     return $O$
```

loop in CheckGuesses. Note that O is already inductive whenever CheckGuesses is called, so CheckGuesses would return O in the worst case.

General quantifier-free and quantified guesses For each CHC C, the algorithm generates initial guesses for an uninterpreted predicate in the head of C based on the templates specified later in Sec. 5.5 and 5.6.1.

After O has been updated based on these guesses, O's interpretations will have captured information-flow summaries for each procedure. If O is a solution for the system of CHCs, then these summaries may be sufficient for proving that the assertions of the program hold. Otherwise, the current procedure summaries are not strong enough for proving that the assertions hold, and the algorithm aims to learn additional property-directed summaries.

Property-directed guesses A *property-directed summary* here refers to a summary that is learned while taking into account the property in some way. All the summaries learned in Chapter 4 can be considered to be property-directed because they all in some way are derived by factoring in the bounded environment. Recall that the bounded environment approximates the full environment containing the body of the query CHC, and that the body of a query CHC specifies a property. To get such property-directed summaries, the InferSum procedure invokes the VerifyWithGuesses procedure, which is a modified version of the Verify procedure from Algorithm 5 in Chapter 4.

The only difference between the VerifyWithGuesses and Verify procedures is that for inferring over-approximate summaries, VerifyWithGuesses will first try to guess over-approximate summaries using templates first, before falling back on the interpolating-solver-based methods described in Chapter 4. Specifically, rather than directly invoking the OOIL or UOIL procedures, it will first invoke the RefineOver procedure shown in Algorithm 10 and then OOIL or UOIL. In the case of OOIL, the argument M for RefineOver will be O and in the case of UOIL, it will be U. Note that falling back on the other methods allows for handling of cases where invariants that are non-relational or otherwise do not fit into the template for guessed invariants are needed to prove a relational property, since such properties may be found by interpolating solvers.

The third template used to generate the guesses returned by GetPDGuess is described later in Sec. 5.6.2. Each property-directed guess in G (where G is GetPDGuess$(M(\mathit{benv}(n)), \mathit{body}, O))$ is such that if it is used as an interpretation for R in query CHC $\mathit{bctx}(n)$ with all the other predicates using their interpretations in M, then the query CHC will be satisfied (i.e., $M(\mathit{benv}(n)) \wedge g(\vec{x})$ will be unsatisfiable for all $g \in G$).

For such a G, $\mathit{makeGuess}(G)(\vec{x})$ can be viewed as an interpolant separating $\mathit{body}(n)$ and $M(\mathit{benv}(n))$ when $\mathit{tgt}(n) = R(\vec{x})$ for some $R \in \mathcal{R}$; to populate G, GetPDGuess generates guesses that obey the syntactic requirements for such an interpolant and adds them to G only after checking that they maintain the invariant that $\mathit{makeGuess}(G)(\vec{x})$ is an interpolant.

Different orders in exploring nodes in VerifyWithGuesses may result in learning different summaries because it will lead to considering different pairs of formulas for which interpolants need to be guessed. However, regardless of the order of exploration, the summaries discovered throughout will constitute a solution for the system of CHCs.

Note that if neither the templates nor the backend interpolating solver can guess the required invariants, the top-level algorithm in Algorithm 9 may not terminate, either because the second top-level loop may never terminate or because the Verify-WithGuesses procedure itself may never return. The algorithm can be terminated early by the user and still return the properties discovered so far, which may be useful for code understanding and can provide hints to the user about manual annotations that may be required. In our experiments (Sect. 5.7), no manual annotations in the benchmark examples were needed to be able to solve them.

Theorem 5.4.1. InferSum *only returns a result* $(\mathit{safe}, (O, U))$ *when* O *is a solution for the system of CHCs* $\mathcal{C}$ *and the original program is safe.*

Proof. This follows directly from the definition of InferSum, since it will not exit the while loop on line 7 of Algorithm 9 unless O is a solution for the set of CHCs. Recall from Chapter 2 that the solution O for $\mathcal{C}$ consists of invariants that are strong enough to show that the assertions in the original program (and captured by the query CHCs) hold, so if a solution O exists, the original program is safe. □

The following theorem shows that the mapping O maintained by InferSum is always inductive, so that checking the condition for the loop on line 7 of Algorithm 9 can be done just by checking whether the query CHCs are satisfied by the current interpretations in O. (Recall from Chapter 2 that a solution for a system of CHCs is an inductive mapping that satisfies all query CHCs in the system.)

Theorem 5.4.2. InferSum *always maintains an inductive mapping* O*.*

Proof. InferSum begins with O being the inductive mapping that maps each n-ary predicate R to $\lambda x_1, \ldots, x_n.\top$. O can be updated only by assigning it to the result of calls to CheckGuesses (either at the top-level of InferSum or within a nested call to RefineOver), by updates made by other parts of the Clover algorithm's OOIL or UOIL procedures, or through updates made within VerifyWithGuesses, which all always result in the updated mapping O being an inductive. It follows that O being inductive is an invariant throughout InferSum. $\square$

Finally, note that the proposed SyGuS approach is not inherently limited to verifying secure information flow or to two copies of a program ($k = 2$). It can be adapted to verify k-hyperproperties for $k > 2$ by extending the basic grammar (shown later in Figure 5.4) to cover target properties. Furthermore, our ideas on property-directed guesses are not specific to information flow and can apply to other properties.

5.5 Grammar Templates without Quantifiers

Figure 5.4 lists the grammar used in the GetQFGuesses procedure (used in Algorithm 9) to generate quantifier-free guesses that represent information-flow properties. Each guess has the form of an implication and corresponds to a relational property because the activation variables `b1` and `b2` always occur positively in the antecedent. The antecedent (*lhs*) allows additional conjuncts expressing equalities (*inEq*) and

$$\begin{aligned}
\mathit{guess} &::= \lambda\vec{x}.\mathit{lhs} \Rightarrow \mathit{rhs} \\
\mathit{lhs} &::= b_1 \wedge b_2 \mid \mathit{inEq} \wedge \mathit{lhs} \mid \mathit{inIneq} \wedge \mathit{lhs} \\
\mathit{rhs} &::= \mathit{outEq} \;\mid\; \mathit{ok} \;\mid\; \mathit{declassify} \\
\mathit{inEq} &::= \mathit{Eq}(\mathit{inArg}) \;\mid\; \mathit{EqArr}(\mathit{inArrArg}, \mathit{ctr}) \\
\mathit{outEq} &::= \mathit{Eq}(\mathit{outArg}) \;\mid\; \mathit{EqArr}(\mathit{outArrArg}, \mathit{ctr}) \\
\mathit{inIneq} &::= c < \mathit{inIntArg} \;\mid\; c \leq \mathit{inIntArg} \;\mid\; c > \mathit{inIntArg} \;\mid\; c \geq \mathit{inIntArg}
\end{aligned}$$

Figure 5.4: Grammar for generating quantifier-free guesses for information flow.

inequalities (*inIneq*) over input arguments of procedures (*inArg*), including arrays (*inArrArg*) indexed by expressions (*ctr*). The consequent (*rhs*) allows conjuncts expressing equalities (*outEq*) over output arguments of procedures (*outArg*, *outArrArg*), the results of assertions (*ok*), or declassify expressions (*declassify*). In the equalities, the expression $\mathit{Eq}(e)$ represents the equality $e = \mathit{idx}(e, 2)$, and $\mathit{EqArr}(e, i)$ represents the equality $e[i] = \mathit{idx}(e[i], 2)$. The inequalities allow comparison of input integer arguments (*inIntArgs*) against constants (c).

The terminals in our grammar are populated from a combination of variable types and a syntactic analysis of the CHC encoding of the body of the target procedure. The candidate variables include input/output parameters of procedure and outputs that store the result of assertions. Various expressions are also extracted from the CHC encoding to serve as terminals in the grammar, e.g., those representing indices in array accesses, or consequents in declassify assertions. Other than activation variables and the results of assertions, all terminals e in the grammar are such that $\mathit{getIdx}(e) = 1$ to reduce redundancy among guesses due to symmetry resulting from indices, e.g., in equality expressions. The complete set of terminals is described below.

Tagging For a CHC with head $R(\vec{x})$ that encodes a modular product procedure `r`, each $x \in \vec{x}$ is tagged as follows:

- `in`: if x corresponds to a non-activation input argument `x` in `r` with $\mathit{getIdx}(\mathtt{x}) = 1$;

- `out`: if x corresponds to an output *ret* in `r` with $\mathit{getIdx}(\mathit{ret}) = 1$;
- `arr`: if x is an array;
- `int`: if x is an integer;
- `ok`: if x is an output value storing the result of assertions.

The following metavariables specify what the terminals based on tags range over:

- *inArg*: the set *inArgs* of variables tagged `in`;
- *inArrArg*: the set *inArrArgs* of variables tagged both `in` and `arr`;
- *outArg*: the set of variables *outArgs* tagged `out`;
- *outArrArg*: the set of variables *outArrArgs* of variables tagged both `out` and `arr`;
- *inIntArg*: the set of variables *inIntArg* tagged `in` and `int`;
- *ok*: the set of variables tagged `ok`.

The activation variables in $\vec{x}$ are denoted b_1 and b_2.

Syntactic Analysis The terminal *ctr* is based on a syntactic analysis of the body of the CHC passed as an argument to GetQFGuesses. It ranges over a set *ctrs* comprising the following:

- all expressions e with $\mathit{getIdx}(e) = 1$ that occur in the procedure body within subexpressions of the form $a[e]$ for some a;
- terminals that c ranges over, consisting of all integer constants that occur as the right- or left-hand side of equalities or inequalities in the body of the procedure;

- terminals that *declassify* ranges over, which consists of the consequents $e_1 = e_2$ of any implications of the form $b_1 \wedge b_2 \Rightarrow e_1 = e_2$, where b_1 and b_2 are Boolean variables, $\mathit{getIdx}(e_1) = 1$, and $\mathit{getIdx}(e_2) = 2$.

5.6 Grammar Templates with Quantifiers

In this section, I present two templates for generating guesses with quantifiers – one for arrays and the other for property-directed invariants.

5.6.1 Quantified Templates for Arrays

This GetQuantifiedGuesses procedure used in Algorithm 9 generates guesses for quantified invariants for a given procedure by adapting a technique from prior work [74] to target *relational* properties. We consider here the task of generating a quantified invariant from a CHC $\mathit{body}(\vec{x}) \Rightarrow R(\vec{x})$. This quantified invariant constitutes a potential over-approximate summary for the procedure encoded by R. A guess for a quantified invariant is constructed from four parts:

- a set of quantified variables qVars not in $\vec{x}$,
- a *range* formula over the variables in $\mathit{inIntArgs} \cup \mathit{qVars}$,
- a set of *equalities* over variables in $\mathit{qVars} \cup \mathit{inIntArgs} \cup \mathit{idx}(\mathit{inIntArgs}, 2)$,
- a *cell property* formula over the variables in $\vec{x} \cup \mathit{qVars}$.

All these components except *equalities* come directly from prior work [74], which combined them to form a candidate invariant: $\forall \mathit{qVars}.\mathit{range} \Rightarrow \mathit{cell\ property}$. The approach for the information flow context is similar but uses *equalities* to guess invariants over both program copies. It also uses activation variables in the antecedent of the implication so that the candidate invariant only applies when both program copies are aligned. Here the template only generates *range* formulas over variables for the first program copy and uses the equalities to ensure that the corresponding variables in the second copy are equal to those in the first.

Quantified variables and range variables are determined similarly to previous work [74]. For each variable `i` in $\mathit{inIntArgs} \cap \mathit{ctrs}$ used to access an array index, two fresh quantified variables `q1` and `q2` are added to qVars, where $\mathit{idx}(\mathtt{q1}, 2) = \mathtt{q2}$. We let $\mathit{quant}(\mathtt{i}) = \mathtt{q1}$. For each such variable, GetQuantifiedGuesses also generates a range formula that is an inductive invariant for R of the form:

$$\mathit{range} ::= \mathtt{i} \leq \mathtt{q1} < \mathit{boundGt} \;\mid\; \mathit{boundLt} < \mathtt{q1} \leq \mathtt{i}$$

Here, *boundGt* is the set of expressions e over variables $\vec{x}$ for which $i < e$ or $e > i$ occurs as a subexpression of *body*, the body of a procedure. Similarly, *boundLt* is the set of expressions e over variables $\vec{x}$ for which $e < i$ or $i > e$ occurs as a subexpression of *body*. Let the set *ranges* denote the set of such *range* expressions that are inductive for R (which is first checked for each such candidate).

For each variable `i` in $\mathit{inIntArgs} \cap \mathit{ctrs}$, GetQuantifiedGuesses generates equality $\mathit{quant}(\mathtt{i}) = \mathit{idx}(\mathit{quant}(\mathtt{i}), 2)$ and equality $i = \mathit{idx}(\mathtt{i}, 2)$ and adds them to the set *equalities*.

Finally, to generate cell properties, GetQuantifiedGuesses considers the subset of expressions generated by the grammar in Figure 5.4 that contain accesses to array cells (also known as *select*-terms and denoted $[\cdot]$) with indices *Idx* such that for each $i \in \mathit{Idx}$, *ranges* contains an expression containing $\mathit{idx}(i, 1)$. It takes each such expression e and substitutes each occurrence of any variable $i \in \mathit{inIntArg} \cap \mathit{ctr}$ with $\mathit{quant}(i)$. It then adds the resulting expression to the set *cellProps*.

For each $\mathit{cellProp} \in \mathit{cellProps}$, we generate the following candidate invariant:

$$\lambda \vec{x}. \forall \mathit{qVars}. \bigwedge \mathit{ranges} \wedge \bigwedge \mathit{equalities} \wedge b_1 \wedge b_2 \Rightarrow \mathit{cellProp}$$

5.6.2 Property-Directed Templates

The final template allows for the generation of property-directed guesses for a particular procedure `r` given a mapping O to inductive interpretations and a mapping M for approximating an environment. This template, used by the GetPDGuess procedure in Algorithm 10 consists of two parts: a *context guess* and a *quantifier-free guess*. As mentioned previously, it aims to find interpolants using SyGuS rather than an interpolating solver. The context guess is used to incorporate relevant properties from the context into the guess, and the quantifier-free guess is used to strengthen it.

I first describe how to generate the context guess for a node n and mapping M. Let $Ands$ be the set of conjuncts in $M(benv(n))$. Each element of the powerset $\mathcal{P}(Ands)$ can become a context guess. We are interested only in elements p in $\mathcal{P}(Ands)$ that represent properties that, while initially not guaranteed to be true whenever `r` is called, are guaranteed to hold for any subsequent recursive calls to `r` provided that they held at the initial invocation of `r`. The technique discovers the largest set $conseqAnds \subseteq \mathcal{P}(Ands)$ that represents such properties through a procedure based on the Houdini algorithm [77] (as shown in the algorithm in Figure 11).

The procedure in Figure 11 examines each CHC in $\mathcal{C}$ with an application of R to variables $\vec{x}$ in its head. The mapping O' maps R to the interpretation $\lambda \vec{y}. \bigwedge Ands$ but is otherwise the same as the current mapping O. For each such CHC, it checks if $O'(R)$ is inductive (line 5) and uses a model cti returned by the backend SMT solver (called a counterexample-to-induction) to weaken $Ands$. It filters out the conjuncts $FalseConj$ of $Ands$ that do not represent properties that are guaranteed to hold for the recursive call. It can now use $\mathcal{P}(conseqAnds)$ as the set of context guesses.

Quantifier-free guesses $QFGuesses$ for $body \Rightarrow R(\vec{x})$ are generated as shown in Sec. 5.5, except now the set c of integer constants also includes all integer constants in $benv(n)$.

Algorithm 11 Procedure to find largest useful element in $\mathcal{P}(Ands)$.

```
procedure FILTER($Ands$, $R(\vec{y})$, $\mathcal{C}$, $O$)
    $O' \leftarrow O[R \mapsto \lambda\vec{y}.\bigwedge Ands]$
    for $body \Rightarrow R(\vec{x}) \in \mathcal{C}$ do
        for application $R(\vec{x}')$ in $body$, context $ctx$ do
            $query \leftarrow O'(R)(\vec{x}) \wedge O'(ctx) \wedge \neg O'(R)(\vec{x}')$
            if $query$ satisfiable then
                $cti \leftarrow$GETMODEL($query$)
                $FC \leftarrow$FALSECONJS($m$, $O'(R)(\vec{x}')$, $Ands$)
                return FILTER($Ands \setminus FC$, $R$, $\mathcal{C}$, $O$)
    return $Ands$
```

Algorithm 12 describes how the bounded environment and quantifier-free guesses are combined to make a guess for a node n with uninterpreted predicate $pr(n) = R$ and the current over-approximate summaries in O. For each $\lambda\vec{x}.lhs \Rightarrow rhs \in QFGuess$ and $p \in \mathcal{P}(conseqAnds)$, COMBINEGUESS considers the mapping $O' = O[R \mapsto \lambda\vec{x}.O[R](\vec{x}) \wedge rhs]$, which is the same as the mapping O except the interpretation for R is updated to $\lambda\vec{x}.O[R](\vec{x}) \wedge rhs$. If $O'(ctx)$ is unsatisfiable and $lhs \wedge p$ is satisfiable (line 5), we generate the following guess: $\lambda\vec{x}.lhs \wedge p \Rightarrow rhs$. We only consider guesses such that $O'(d \wedge benv(n))$ is unsatisfiable because these guesses are the only ones that can serve as an interpolant between d and $benv(n)$. The d here is a disjunction contained within $body(n)$, so serving as an interpolant between d and $benv(n)$ is a prerequisite for serving as an interpolant between $body(n)$ and $benv(n)$. This requirement ensures that the guesses considered help make progress toward updating O so that $O[R]$ is an interpolant for $body(n)$ and $benv(n)$. This in turn ensures that progress is made toward proving the assertion in the original program corresponding to query Q.

The checks on line 5 guarantee that each element added to $Guesses$, when applied to $\vec{x}$, is an interpolant separating $O'(d)$ and $O'(benv(n))$. If all guesses in $Guesses$, are interpolants separating these formulas when applied to $\vec{x}$, then it follows that $makeGuess(Guesses)(\vec{x})$ is also such an interpolant. Note that these guesses may contain quantifiers if the interpretations in O contain quantifiers.

Algorithm 12 Inference procedure for property-directed guesses.

```
1: procedure CombineGuess($QFGuess$, $conseqAnds$, $O$, $n$, $d$)
2:     for $\lambda\vec{x}.lhs \Rightarrow rhs \in QFGuess$ do
3:         for $p \in \mathcal{P}(conseqAnds)$ do
4:             $O' \leftarrow O[R \mapsto \lambda\vec{x}.O(R)(\vec{x}) \wedge rhs]$
5:             if $O'(d \wedge benv(n))$ unsat, $lhs \wedge p$ sat then
6:                 $Guesses \leftarrow Guesses \cup \{\lambda\vec{x}.lhs \wedge p \Rightarrow rhs\}$
```

5.7 Implementation and Evaluation

I have implemented this technique in a tool called FLOWER, developed on top of the CHC solver FREQHORN [72, 73]. I evaluated it on a suite of benchmarks[1] from the literature and real-world examples.

In the implementation, all candidate guesses allowed by the grammars are enumerated and checked, i.e., there is no further heuristic selection (currently) in the tool. Although this lack of heuristic selection can be problematic if there are too many guesses, the tool did not encounter this issue in practice. For property-directed guesses, the unfoldings are explored in a breadth-first like manner.

Benchmarks Of the 29 benchmarks, 15 are based on a subset of the evaluation set for MPPs [66, 13, 20, 50, 25, 56, 79, 141, 148]. The subset was obtained by leaving out termination-related properties, since automating the verification of these would require the synthesis of ranking functions, which is not supported by FLOWER. While small in size, with the original programs ranging from 24-70 lines of VIPER [116] code, these programs include non-trivial features such as arrays and declassification that are challenging for automated verifiers. I added two benchmarks based on code from Amazon Web Service's S2N [11], about 160 lines of SMT-LIB2 code that involve reading/writing from buffers. We also translated six benchmarks based on BLAZER's "Literature" and "STAC" benchmarks [13], which ranged from 41-208 lines of Java.

[1]Available at https://github.com/lmpick/flower-benchmarks

The Viper benchmarks contained many manual annotations of information-flow specifications for both procedures and loops. In this evaluation, I treated the specification for the apparent top-level procedure as an assertion and eliminated the remaining annotations. Loops were encoded as recursion, as is typical in CHC encodings. Memory locations and memory-related annotations in the benchmarks were not encoded in CHCs; structures were either flattened or encoded as arrays.

The Blazer benchmarks considered were written in Java and originally checked for timing side channels. Checking for timing side channels can be reduced to checking for noninterference with appropriate instrumentation [17]. I manually instrumented these benchmarks in this way and encoded them into CHCs.

Evaluation I also compared the tool against Descartes [143] and Spacer [103]. For Descartes, I translated the CHC benchmarks into intraprocedural Java programs.

Results from experiments on the suite of 29 benchmarks with a timeout of 10 minutes are shown in Table 5.1. Blazer benchmarks are prefixed with "B" and s2n benchmarks are prefixed with "s2n." A timeout is indicated with **TO** and an unknown result with **U**. N/A indicates that Descartes was unable to handle the benchmark because of the presence of arrays or declassification. Benchmarks were run on a MacBook Pro, with a 2.7GHz Intel Core i5 processor and 8GB RAM.

Flower is able to solve all 29 benchmarks, including all 15 benchmarks originally used to assess the usefulness of MPPs. Note that Flower successfully solved all these examples *without the annotations required by* Viper[116]. These results demonstrate the effectiveness of the presented approach in reducing the annotation burden for verifying secure information flow.

Spacer is able to solve 14 of the 29 benchmarks, timing out for 14, and reporting **U** for one. Descartes cannot handle the majority of the benchmarks; of the 10 benchmarks it can take as input, Descartes solves 5. Out of the 20 examples with recursion (marked in Column 2), Spacer can only solve 5, whereas our tool can handle all 20. Spacer finds invariants via interpolation, which is unlikely to directly capture relational properties, so it is unable to find suitable invariants for these recursive procedures. For recursion-free examples, relational invariants are less crucial; invariants capturing precise behaviors are easier to find and are often sufficient for verification.

Descartes is similarly unable to find appropriate invariants. For each of the 5 recursive benchmarks that it can take as input, it is unable to find the required loop invariant to verify the program. Although Descartes also uses a template-based approach for generating candidate invariants, the templates are insufficient for these benchmarks.

To evaluate scalability, I considered multiple versions of the Costanzo benchmark with different array bounds (shown in parentheses in Table 5.1). Figure 5.5 shows the performance comparison against Spacer as the array bound increases. Spacer's behavior indicates its inability to find relational properties; it learns properties for each array index individually, rendering it unable to solve the Costanzo benchmark within 10 minutes after the array bound reaches 16 (note that the original Costanzo benchmark has bound 64). Although it was run in a mode that allows it to learn quantified properties, Spacer is unable to find the desired relational property. In contrast, our approach solves all the bounded Costanzo benchmarks in about the same time because the quantified guesses are the same except for the constant bound. Our approach is also able to solve the Costanzo benchmark in which the array is *unbounded*, which Spacer is unable to do.

Table 5.1: Results for 29 benchmarks. Times shown in seconds.

Example	Recursive	Flower Time	Spacer Time	Descartes Time
Banerjee		8.00	0.04	N/A
B GPT14	✓	73.91	**TO**	**U**
B K96	✓	12.60	**TO**	**U**
B Login	✓	18.20	**TO**	N/A
B ModPow1	✓	60.86	**TO**	**U**
B ModPow2	✓	104.59	**TO**	**U**
B PWCheck	✓	18.04	**TO**	N/A
Costanzo (2)	✓	3.94	0.65	N/A
Costanzo (4)	✓	3.85	7.10	N/A
Costanzo (8)	✓	3.85	62.50	N/A
Costanzo (16)	✓	4.08	**TO**	N/A
Costanzo (32)	✓	3.88	**TO**	N/A
Costanzo (64)	✓	3.93	**TO**	N/A
Costanzo (unbounded)	✓	8.17	**TO**	N/A
Darvas		2.04	0.03	N/A
Declassification	✓	4.91	0.03	N/A
Joana Fig. 1 top left		0.96	0.03	N/A
Joana Fig. 2 bottom left		0.90	0.02	0.06
Joana Fig. 2 top		0.58	0.02	0.08
Joana Fig. 13 left		0.25	0.03	0.07
Kusters		8.07	0.03	0.09
Main Example	✓	135.90	**U**	N/A
Main Example (det.)	✓	13.98	**TO**	N/A
s2n Ex. 1	✓	352.70	0.06	N/A
s2n Ex. 2	✓	30.95	**TO**	N/A
Smith	✓	23.26	**TO**	N/A
Terauchi Fig. 1		0.40	0.03	0.08
Terauchi Fig. 2		0.84	0.03	N/A
Terauchi Fig. 3	✓	3.55	**TO**	**U**

5.8 Related Work

There are many related efforts in relational and hyperproperty verification, CHC solving, information-flow checkers, and syntax-guided synthesis. A large body of the related work in relational and hyperproperty verification has been described in Sect. 3.5, so in this section I will highlight the aspects that are relevant for the work presented in this chapter. For related work on CHC solving, see Sect. 4.9.

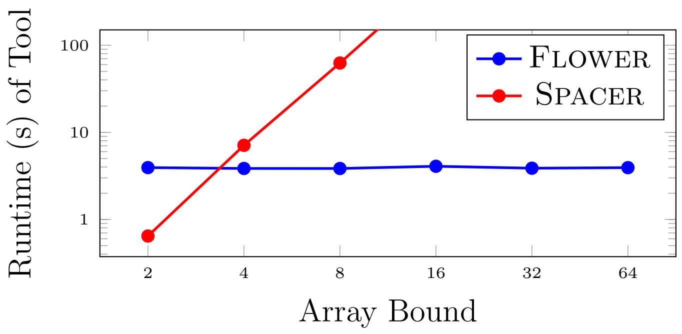


Figure 5.5: Timing results for Costanzo benchmark with different array bounds.

5.8.1 Relational Program Verification

While this work focuses on modular product programs [66], many other approaches also reduce relational program verification to safety verification [148, 28, 27, 25, 26, 89, 48], including those that employ a reduction to systems of CHCs [115, 59]. *However, most do not perform modular reasoning over procedures but inline them, and do not generate relational specifications for procedures.*

Other than the work on MPPs [66], which works only for hyperproperties, there exist few other approaches that allow for automated modular reasoning over relational programs. One such approach restricts both copies of the program to always follow the same control flow [25]. Another such approach uses *mutual* summaries to capture relational specifications of procedures [89], but similarly to work on MPPs, this work also does not provide an automatic procedure for inferring summaries. There is one synchronization approach that uses property-directed reachability and uses modular reasoning for inference of relational procedure summaries [103, 93, 115], but experiments in Sect. 5.7 show that SPACER alone often fails to infer the needed invariants in programs with recursion. This dearth of techniques for inferring modular relational procedure summaries motivated the work on FLOWER described in this chapter.

5.8.2 Information-Flow Properties and Verification

Most automatic hyperproperty verifiers can handle information-flow properties by constructing product programs either implicitly [143, 70] or explicitly [25, 26, 66], or by lazily performing self-composition [139, 155] or synchronization [115, 59], as mentioned above.

Other efforts focus on quantitative information flow, where the aim is to verify *resource* leakage, such as the presence of timing side channels [13, 41, 17]. With appropriate instrumentation for resource leakage [17, 41], checking for timing leakage can be reduced to hyperproperty verification. In particular, the absence of timing side channels can be reduced to checking for non-interference after appropriate resource usage instrumentation, allowing tools for checking non-interference to check for the absence of timing side channels as well, as demonstrated in the evaluation of FLOWER in Sect. 5.7.

Approaches based on types and abstract interpretation can modularly infer information-flow properties of procedures. There are many type-inference-based approaches for checking secure information flow [62, 151, 117, 122, 20, 146], many of which are aware of programming-language specific features such as objects [122, 117, 20, 146] or are even modular with respect to program structure [146]. Such approaches employ a security type system such that terms only type check if they do not have any illegal information flows (e.g., from low-security to high-security variables). There are also approaches based on dynamic taint analysis [133, 49, 53, 99, 135, 142], which involves instrumenting code with taint variables and code to track taint. However, type-inference-based and taint analysis approaches suffer from imprecision (e.g., due to path-insensitivity or an inability to infer invariants over arrays) that may lead to failure in type inference even for leakage-free programs. In contrast, our approach is path-sensitive and requires only the annotations that specify the property to be verified. One abstract-interpretation-based approach can infer possible information-flow dependencies, indicating which variables' values may depend on others' [156]. This approach, like ours, does not require annotations indicating which inputs and outputs are public or private. However, unlike our approach, it does not handle programs with procedures, arrays, or declassification. A more recent abstract interpretation approach [16] proposes hypercollecting semantics, which it uses to derive an analysis for quantitative information flow.

5.8.3 Syntax-Guided Synthesis

The presented approach for inferring information flow specifications is also related to a wide range of guess-and-check SyGuS techniques [8, 120, 10, 72, 73, 129, 74]. Especially relevant are enumerate-and-check approaches to solve CHCs [72, 73, 74]. The template for guessing quantified invariants for arrays in Chapter 5 adapts a previous technique [74] to the setting of reasoning about secure information flow. Such techniques have not previously been applied to inferring or verifying information-flow properties. The structure of information-flow properties makes them ideal targets for grammar-based enumerative search and synthesis and motivated the work presented in Chapter 5.

Chapter 6

Conclusions and Future Work

I have described to apply and improve automated modular verification of programs in order to achieve scalability by taking advantage of the syntax and structure of properties and programs. In particular, I have demonstrated how to do this for k-safety of intraprocedural programs, where modularity stems from the ability to decompose a verification problem into several subproblems when control-flow branches (such as in the case when there are loops), and for interprocedural programs, where modularity can also result from using procedure boundaries to formulate verification subproblems. Finally, I have presented work considering information-flow verification of interprocedural programs, which is a specific domain within the intersection of k-safety verification and interprocedural verification.

6.1 Conclusions

In both relational and non-relational settings, awareness of useful syntactic qualities of properties allowed for significant improvements over previous state-of-the-art techniques that were largely agnostic to these particular features of properties. Specifically, the k-safety verifier Synonym was able to take advantage of the symmetries of properties to handle larger examples than the Descartes tool upon which it was built, Clover's use of EC lemmas allowed it to solve benchmarks containing mutual recursion that other tools were unable to handle, and Flower was able to use grammar templates to infer invariants that allowed it solve unannotated benchmarks that other tools could not.

The awareness of the syntactic and structural features of programs can be seen largely as serving two purposes in my work: (1) being useful for guiding the formulation of verification subproblems so that solving these subproblems involves learning or considering intermediate properties that demonstrate desired syntactic features, or (2) being involved in determining what intermediate properties are desirable for discharging intermediate verification problems. In other words, it is the desire to find and consider invariants and intermediate properties with these features that motivates the consideration of the syntax and structure of the programs.

The first purpose can be seen in each of the three kinds of verification problems considered. In k-safety verification, synchronization aims to align matching fragments of program copies, making it more likely that a simpler relational invariant can be learned. In interprocedural program verification, verification algorithms use the modular structure of interprocedural programs to learn invariant properties for procedures. The presented work on information-flow property verification counts both as k-safety verification and interprocedural program verification, where synchronization is used to align structural features of program copies and the modular structure of interprocedural programs is used to generate verification subproblems. The goal of such alignments is to provide opportunities for the inference of relational invariants that exhibit the features expressed in the grammar: equalities of corresponding variables across program copies, equalities of corresponding array elements across program copies, or assumptions about the environment in which a procedure is called (for handling declassification).

The second purpose can also be seen in each kind of verification problem considered. The work on symmetry in Chapter 3 relied on the symmetries found in the properties and used those to perform reductions, eliminating redundant verification tasks, but these symmetries must apply to the composed programs. In other words, the ideal property is one that exhibits symmetries that are also symmetries for the corresponding programs. The algorithm used in SYNONYM detects when such properties arise during verification and uses them to discharge redundant verification tasks. The formulation of EC lemmas for interprocedural program verification more directly involves program structure since the definition of an EC lemma involves considering the program call graph. These EC lemmas are used to discharge verification subproblems that involve mutually recursive procedures. Finally, this second purpose can be seen in the grammar templates for guessing invariants presented in Chapter 5. These grammar templates incorporate program syntax so that the guessed invariants are influenced by the syntactic features of the program. FLOWER uses these invariants to solve per-procedure verification subproblems for information-flow properties.

This general approach, where syntactic and structural features of a program are used to help set up verification subproblems involving desirable intermediate properties, is likely to be useful for other kinds of verification problems beyond the ones considered in this dissertation as well. Both identifying what features are useful to have in intermediate properties and determining how to use program structure in an algorithm to achieve such intermediate properties are applications of human insights. As has been seen in the work presented in this dissertation, encoding these insights into an algorithm can help scale verification or enable verification for programs and properties that otherwise cannot be handled well by more general automated reasoning techniques.

6.2 Future Work

6.2.1 Heuristic Improvements

As mentioned in Chapter 4, several heuristics are used in Clover (note that these are also present in Flower, since it is implemented on top of Clover). In particular, the heuristics used to prioritize the choice of which node to process next lead to a BFS-like search where in some cases a DFS-like search may perform better. Investigating better heuristics for Clover's search remains future work.

Clover and Flower also include heuristics to ensure nontrivial over-approximate summaries are used even when the backend interpolating solver returns an interpolant that is simply $\top$. One other avenue for improving the over-approximate summaries in practice is using different backends to provide interpolants, including tools like HornSpec that provide nontrivial solutions [131]. Future work includes investigating the effect of such solvers on the performance of Clover and Flower.

6.2.2 Symmetry-Breaking in CHC Solving

The work presented in Chapter 5 directly incorporates and builds upon the work presented in Chapter 4, with the template-based summary inference taking place before and within the modular verification algorithm presented in Chapter 4. The corresponding tool Flower too was correspondingly built upon the Clover tool described in Chapter 4. The tool Flower, however, does not benefit from the symmetry-breaking techniques proposed in Chapter 3 as they were implemented in a different tool. Future work includes applying the symmetry-breaking techniques proposed in Chapter 3 to a CHC-solving setting and implementing this symmetry-breaking in the CHC solver Flower.

It is likely that Flower will exhibit different benefits than Descartes did from symmetry breaking because of the difference in the exploration of program behaviors and form of invariants, and comparing the performance of Synonym and Flower with symmetry-breaking would be an additional direction to explore in future work. In particular, this future work should measure whether the exploration and invariants used in Flower create more opportunities to apply symmetry-breaking.

Descartes and Synonym are based on Hoare logic and employ a forward exploration strategy; furthermore, they perform alignment dynamically during exploration and focus on aligning loops across program copies, but these loops may not correspond to copies of the same loop. Meanwhile, Flower's exploration, like most CHC solvers, is neither strictly forward nor backward, and, more crucially, it relies on a MPP construction, which guarantees that corresponding procedures and loops across program copies – and *only* corresponding ones – have the opportunity to be aligned, regardless of intermediate invariants. This alignment of corresponding program structures across copies may create more opportunities for symmetry-breaking to be applied.

The difference in the form of invariants that Flower infers may more critically impact the application of symmetry-breaking. Most of the intermediate invariants inferred by Descartes and Synonym are the result of a strongest postcondition computation, with the only exception being that loop invariants are inferred using SyGuS with a simple grammar template for inferring relational properties. Meanwhile, the majority of intermediate invariants inferred by Flower are generated by grammar templates for inferring relational properties. Because these grammar templates are designed to exhibit symmetries, they may lead to more symmetries in invariants than would be present in Descartes or Synonym, presenting more opportunities to identify and eliminate redundant verification subtasks.

6.2.3 Heap Properties

As touched upon briefly in Sect. 3.5, CHC-based verifiers have not been able to encode heap properties easily; most CHC encodings of programs that have heap data structures either represent them by using the theory of arrays to represent heap contents or transform the heap data structures away using invariants [68]. More recently, there has been a proposal exploring the possibility of extending the SMT-LIB format for CHCs with a theory of heaps, allowing for CHC-based verifiers to operate on heap structures directly [68]. As can be seen in the work on relational program logics that incorporate aspects of separation logic to reason about the heap [152, 21], dealing with heaps in a relational setting is nontrivial, and aligning heap data structures for synchrony is typically a manual effort, since there exist many possible alignments.

Relevant to the automatic handling of heaps are promising SyGuS-based methods for inferring invariants about algebraic datatypes [154, 71], including recent work in a relational setting for proving equivalence [71]. Generalizing these insights and applying them to future work in CHC-based k-safety verification will include trying to automate finding heap object alignments using heuristics, which, in a similar vein to the approaches presented in this dissertation, will include exploiting structural properties of the heap representation at intermediate points during CHC solving.

Bibliography

[1] The Agda wiki. https://wiki.portal.chalmers.se/agda/, 2021. Accessed: 2021-08-24.

[2] The Coq proof assistant. https://coq.inria.fr/, 2021. Accessed: 2021-08-24.

[3] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. A relational logic for higher-order programs. *Proceedings of the ACM on Programming Languages*, 1(ICFP):21:1–21:29, 2017.

[4] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 789–801. ACM, 2016.

[5] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From under-approximations to over-approximations and back. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2012.

[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 53–70. USENIX Association, 2016.

[7] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, 2006.

[8] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in*

Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, pages 1–8. IEEE, 2013.

[9] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 98–109. ACM, 2005.

[10] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336, 2017.

[11] Amazon Web Services. s2n. https://github.com/awslabs/s2n. Accessed: 2021-08-24.

[12] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 4–16. ACM, 2002.

[13] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 362–375. ACM, 2017.

[14] Kazuyuki Asada, Ryosuke Sato, and Naoki Kobayashi. Verifying relational properties of functional programs by first-order refinement. *Science of Computer Programming*, 137:2–62, 2017.

[15] Sepideh Asadi, Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen, Karine Even-Mendoza, Natasha Sharygina, and Hana Chockler. Function summarization modulo theories. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 56–75. EasyChair, 2018.

[16] Mounir Assaf, David A. Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. Hypercollecting semantics and its application to static analysis of information flow. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming*

Languages, POPL 2017, Paris, France, January 18-20, 2017, pages 874–887. ACM, 2017.

[17] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. SideTrail: Verifying time-balancing of cryptosystems. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers*, volume 11294 of *Lecture Notes in Computer Science*, pages 215–228. Springer, 2018.

[18] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In John Field and Gregor Snelting, editors, *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pages 97–103. ACM, 2001.

[19] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[20] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada*, page 253. IEEE Computer Society, 2002.

[21] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. Relational logic with framing and hypotheses. In Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen, editors, *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*, volume 65 of *LIPIcs*, pages 11:1–11:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[22] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[23] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[24] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[25] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael J. Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011.

[26] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*, volume 7734 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2013.

[27] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Product programs and relational program logics. *J. Log. Algebraic Methods Program.*, 85(5):847–859, 2016.

[28] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.

[29] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 97–110. ACM, 2012.

[30] Nels E. Beckman and Aditya V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 211–221. ACM, 2011.

[31] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 14–25. ACM, 2004.

[32] Lennart Beringer. Relational decomposition. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The*

Netherlands, August 22-25, 2011. Proceedings, volume 6898 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2011.

[33] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2007.

[34] Nikolaj Bjørner and Mikolás Janota. Playing with quantified satisfaction. In Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov, editors, *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015.

[35] Sam Blackshear and Shuvendu K. Lahiri. Almost-correct specifications: a modular semantic framework for assigning confidence to warnings. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 209–218. ACM, 2013.

[36] Aaron R. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[37] Aaron R Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Science & Business Media, 2007.

[38] Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. Higher-order constrained horn clauses for verification. *Proceedings of the ACM on Programming Languages*, 2(POPL):11:1–11:28, 2018.

[39] Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. HoIce: An ice-based non-linear horn clause solver. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 146–156. Springer, 2018.

[40] CHC-Comp. `https://chc-comp.github.io`, 2021. Accessed: 2021-08-24.

[41] Jia Chen, Yu Feng, and Isil Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017*

ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, pages 875–890. ACM, 2017.

[42] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. Relational verification using reinforcement learning. *Proc. ACM Program. Lang.*, 3(OOPSLA):141:1–141:30, 2019.

[43] Yu-Fang Chen, Chiao Hsieh, Ming-Hsien Tsai, Bow-Yaw Wang, and Farn Wang. Verifying recursive programs using intraprocedural analyzers. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2014.

[44] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1027–1040. ACM, 2019.

[45] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.

[46] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 450–462. Springer, 1993.

[47] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[48] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[49] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony I. T. Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 133–147. ACM, 2005.

[50] David Costanzo and Zhong Shao. A separation logic for enforcing declarative information flow control policies. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 179–198. Springer, 2014.

[51] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[52] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 128–148. Springer, 2013.

[53] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th Annual International Symposium on Microarchitecture (MICRO-37 2004), 4-8 December 2004, Portland, OR, USA*, pages 221–232. IEEE Computer Society, 2004.

[54] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996*, pages 148–159. Morgan Kaufmann, 1996.

[55] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[56] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2005.

[57] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part*

I, volume 9206 of *Lecture Notes in Computer Science*, pages 324–342. Springer, 2015.

[58] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. VeriMAP: A tool for verifying programs through transformations. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 568–574. Springer, 2014.

[59] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Relational verification through horn clause transformation. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 147–169. Springer, 2016.

[60] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Predicate pairing for program verification. *Theory and Practice of Logic Programs*, 18(2):126–166, 2018.

[61] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[62] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[63] Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, and Andreas Podelski. Ultimate TreeAutomizer (CHC-COMP tool description). In Emanuele De Angelis, Grigory Fedyukovich, Nikos Tzevelekos, and Mattias Ulbrich, editors, *Proceedings of the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR@ETAPS 2019, Prague, Czech Republic, 6-7th April 2019*, volume 296 of *EPTCS*, pages 42–47, 2019.

[64] Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 356–371. Springer, 2011.

[65] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134. FMCAD Inc., 2011.

[66] Marco Eilers, Peter Müller, and Samuel Hitz. Modular product programs. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 502–529. Springer, 2018.

[67] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 1993.

[68] Zafer Esen and Philipp Rümmer. Abstract: Towards an SMT-LIB theory of heap. In François Bobot and Tjark Weber, editors, *Proceedings of the 18th International Workshop on Satisfiability Modulo Theories co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Online (initially located in Paris, France), July 5-6, 2020*, volume 2854 of *CEUR Workshop Proceedings*, page 60. CEUR-WS.org, 2020.

[69] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2010.

[70] Azadeh Farzan and Anthony Vandikas. Automated hypersafety verification. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 200–218. Springer, 2019.

[71] Grigory Fedyukovich and Gidon Ernst. Bridging arrays and adts in recursive proofs. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 24–42. Springer, 2021.

[72] Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. Sampling invariants from frequency distributions. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 100–107. IEEE, 2017.

[73] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Solving constrained horn clauses using syntax and data. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[74] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 259–277. Springer, 2019.

[75] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 349–360. ACM, 2014.

[76] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.

[77] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[78] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 69–87. Springer, 2014.

[79] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, 2015.

[80] Patrice Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 476–479. Springer, 1997.

[81] Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 2001.

[82] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In 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*, pages 43–56. ACM, 2010.

[83] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.

[84] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 405–416. ACM, 2012.

[85] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[86] Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A software model-checker for verification and refutation. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 170–174. Springer, 2006.

[87] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving safety and liveness of practical distributed systems. *Communications of the ACM*, 60(7):83–92, 2017.

[88] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 165–181. USENIX Association, 2014.

[89] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. Towards modularly comparing programs using automated theorem provers. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 282–299. Springer, 2013.

[90] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2009.

[91] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 31–40. ACM, 2005.

[92] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[93] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.

[94] Douglas R Hofstadter et al. *Gödel, Escher, Bach: an eternal golden braid*, volume 20. Basic books New York, 1979.

[95] Hossein Hojjat and Philipp Rümmer. The ELDARICA horn solver. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–7. IEEE, 2018.

[96] C. Norris Ip and David L. Dill. Verifying systems with replicated components in Mur*phi*. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, volume 1102 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1996.

[97] Franjo Ivancic, Gogul Balakrishnan, Aarti Gupta, Sriram Sankaranarayanan, Naoto Maeda, Hiroki Tokuoka, Takashi Imoto, and Yoshiaki Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 133–142. IEEE Computer Society, 2011.

[98] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009.

[99] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[100] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. Relational program reasoning using compiler IR. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*, volume 9971 of *Lecture Notes in Computer Science*, pages 149–165, 2016.

[101] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[102] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2014.

[103] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[104] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 345–355. ACM, 2013.

[105] Akash Lal and Shaz Qadeer. A program transformation for faster goal-directed search. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 147–154. IEEE, 2014.

[106] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 427–443. Springer, 2012.

[107] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[108] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 75–86. ACM, 2009.

[109] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[110] Kenneth L. McMillan. Lazy annotation for program testing and verification. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2010.

[111] Kenneth L. McMillan. Lazy annotation revisited. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2014.

[112] Kenneth L McMillan and Andrey Rybalchenko. Solving constrained horn clauses using interpolation. Technical report, MSR-TR-2013-6, 2013.

[113] Dmitry Mordvinov and Grigory Fedyukovich. Synchronizing constrained horn clauses. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 338–355. EasyChair, 2017.

[114] Dmitry Mordvinov and Grigory Fedyukovich. Verifying safety of functional programs with Rosette/Unbound. *CoRR*, abs/1704.04558, 2017.

[115] Dmitry Mordvinov and Grigory Fedyukovich. Property directed inference of relational invariants. In Clark W. Barrett and Jin Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 152–160. IEEE, 2019.

[116] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

[117] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 228–241. ACM, 1999.

[118] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[119] OCaml Tutorials. `https://ocaml.org/learn/tutorials/labels.html`, 2021. Accessed: 2021-08-24.

[120] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. Data-driven precondition inference with learned features. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 42–56. ACM, 2016.

[121] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630. ACM, 2016.

[122] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems*, 37(2):6:1–6:50, 2015.

[123] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. Exploiting synchrony and symmetry in relational verification. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 164–182. Springer, 2018.

[124] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. Automating modular verification of secure information flow. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 158–168. IEEE, 2020.

[125] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. Unbounded procedure summaries from bounded environments. In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, pages 291–324. Springer, 2021.

[126] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 123–134. ACM, 2007.

[127] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *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. ACM Press, 1995.

[128] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *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. ACM Press, 1995.

[129] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2019.

[130] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2013.

[131] Sumanth Prabhu S, Grigory Fedyukovich, Kumar Madhukar, and Deepak D'Souza. Specification synthesis with constrained horn clauses. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pages 1203–1217. ACM, 2021.

[132] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*, pages 255–269. IEEE Computer Society, 2005.

[133] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kâafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In Pierangela Samarati, editor, *SECRYPT 2013 - Proceedings of the 10th International Conference on Security and Cryptography, Reykjavík, Iceland, 29-31 July, 2013*, pages 461–468. SciTePress, 2013.

[134] Yuki Satake, Hiroshi Unno, and Hinata Yanagi. Probabilistic inference for predicate constraint satisfaction. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1644–1651. AAAI Press, 2020.

[135] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331. IEEE Computer Society, 2010.

[136] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233, 1981.

[137] Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis.* New York University. Courant Institute of Mathematical Sciences, 1978.

[138] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2014.

[139] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Property directed self composition. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 161–179. Springer, 2019.

[140] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In David S. Rosenblum and

Sebastian G. Elbaum, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, pages 174–184. ACM, 2007.

[141] Geoffrey Smith. Principles of secure information flow analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer, 2007.

[142] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *ICISS*, volume 5352 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2008.

[143] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 57–69. ACM, 2016.

[144] Ofer Strichman and Maor Veitsman. Regression verification for unbalanced recursive functions. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 645–658, 2016.

[145] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 367–373. Springer, 2014.

[146] Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2004.

[147] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

[148] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

[149] Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. Automating induction for solving horn clauses. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 571–591. Springer, 2017.

[150] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for Haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014.

[151] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2/3):167–188, 1996.

[152] Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

[153] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 282–291. ACM, 2006.

[154] Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. Lemma synthesis for automating induction over algebraic data types. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 600–617. Springer, 2019.

[155] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. Lazy self-composition for security verification. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 136–156. Springer, 2018.

[156] Matteo Zanioli and Agostino Cortesi. Information leakage analysis by abstract interpretation. In Ivana Cerná, Tibor Gyimóthy, Juraj Hromkovic, Keith G. Jeffery, Rastislav Královic, Marko Vukolic, and Stefan Wolf, editors, *SOFSEM*

2011: Theory and Practice of Computer Science - 37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 22-28, 2011. Proceedings, volume 6543 of *Lecture Notes in Computer Science*, pages 545–557. Springer, 2011.

[157] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 707–721. ACM, 2018.