Synthesizing Software Verifiers from Proof Rules

Sergey Grebenshchikov
Technische Universität München
grebensh@cs.tum.edu

Nuno P. Lopes
INESC-ID / IST - TU Lisbon
nuno.lopes@ist.utl.pt

Corneliu Popeea
Technische Universität München
popeea@model.in.tum.de

Andrey Rybalchenko
Technische Universität München
rybal@in.tum.de

Abstract

Automatically generated tools can significantly improve programmer productivity. For example, parsers and dataflow analyzers can be automatically generated from declarative specifications in the form of grammars, which tremendously simplifies the task of implementing a compiler. In this paper, we present a method for the automatic synthesis of software verification tools. Our synthesis procedure takes as input a description of the employed proof rule, e.g., program safety checking via inductive invariants, and produces a tool that automatically discovers the auxiliary assertions required by the proof rule, e.g., inductive loop invariants and procedure summaries. We rely on a (standard) representation of proof rules using recursive equations over the auxiliary assertions. The discovery of auxiliary assertions, i.e., solving the equations, is based on an iterative process that extrapolates solutions obtained for finitary unrollings of equations. We show how our method synthesizes automatic safety and liveness verifiers for programs with procedures, multi-threaded programs, and functional programs. Our experimental comparison of the resulting verifiers with existing state-of-the-art verification tools confirms the practicality of the approach.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.4.5 [*Operating Systems*]: Reliability—Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Proof rules, verification tool synthesis, software verification, software model checking

1. Introduction

Developing tools that deal with programs, e.g., parsers, compilers, analyzers, or verifiers, is a difficult yet necessary task for increasing programmer productivity. Programs are complex artifacts and their treatment within a tool requires careful consideration of various intricate aspects of program syntax and semantics. Tool synthesis offers an attractive alternative to manual tool development. Once a tool generator is developed for a given problem domain, it can be used to synthesize various tools that deal with particular problem instances in the given domain. For example, in the problem domain of parsing, a parser generator allows one to synthesize parsers for particular (programming) languages [1]. Each parser is synthesized by the generator from a language specification in the form of a grammar. Furthermore, static analyzers (including pointer alias analyzers) can be generated from attribute grammars [47], dataflow equations [29, 31], equations in the form of set constraints [2, 3, 27], or equations as Datalog rules [28, 34, 51]. These approaches take an analysis specification as a set of equations and produce an analyzer that infers program properties by solving the equations.

Recently, property verifiers became a target for automated tool construction. For example, the GETAFIX tool for checking Boolean programs was synthesized from a logical formula in μ-calculus using a BDD-based fixpoint solver [49]. Still, software verification tools such as SLAM [4], BLAST [20, 21], FSOFT [23], IMPACT [33], TERMINATOR [11], CPACHECKER [7], or DSOLVE [24, 43], are developed from the ground up in a complex manual effort that takes into account particularities of domain specific reasoning for the applied verification method. The development efforts may need to be repeated to a large extent once a different proof rule is employed, or programs in a different programming language are considered as input. This status quo hinders the advancement of the state-of-the-art in software verification, makes it prohibitively expensive to develop new verification methods and provide verification tools for the growing number of application domains. Without automatic tool support for the development of software verifiers, we cannot deliver required tools for improving software reliability and leave software developers alone in dealing with the increasing complexity of modern software systems.

In this paper we present a method for automating the development of software verification tools by providing the following key ingredients: a methodology for describing verification methods for reachability and termination properties as constraint solving problems, and an efficient solver for the resulting constraints. As a result, developing a new software verifier will become a two-step process: i) design and specification of a verification method in the form of constraints supported by our methodology, and ii) construction of a frontend that generates constraints from software source code. The main effort will be spent in the formulation of a suitable verification method, which is a creative activity that usually leverages existing methods and adapts them to new application domains. The frontend construction usually requires writing a translator from the compiler's intermediate representation into the language of constraints, which is a well-established routine. We be-

PLDI'12, June 11–16, 2012, Beijing, China.

```
   int sum(int n) {
     int s;
1:   if (n > 0) {
2:     s = sum(n-1);
3:     return s+n;
     } else {
4:     return 0;
     }
5: }
```

$$
\begin{aligned}
V &= (\mathtt{n}, \mathtt{s}, \mathtt{ret}, \mathtt{pc}) \\
\mathit{init}(V) &= (\mathtt{pc} = \ell_1) \\
\rho(V, V') &= (\mathtt{n} \geq 1 \wedge \mathit{move}(\ell_1, \ell_2) \wedge \mathit{skip}(\mathtt{n}, \mathtt{s}, \mathtt{ret})) \vee \\
&\quad (\mathtt{ret}' = \mathtt{s} + \mathtt{n} \wedge \mathit{move}(\ell_3, \ell_5) \wedge \mathit{skip}(\mathtt{n}, \mathtt{s})) \vee \\
&\quad (\mathtt{n} \leq 0 \wedge \mathit{move}(\ell_1, \ell_4) \wedge \mathit{skip}(\mathtt{n}, \mathtt{s}, \mathtt{ret})) \vee \\
&\quad (\mathtt{ret}' = 0 \wedge \mathit{move}(\ell_4, \ell_5) \wedge \mathit{skip}(\mathtt{n}, \mathtt{s})) \\
\mathit{call}(V, V') &= (\mathtt{n}' = \mathtt{n} - 1 \wedge \mathit{move}(\ell_2, \ell_1)) \\
\mathit{ret}(V, V') &= (\mathtt{s}' = \mathtt{ret} \wedge \mathit{move}(\ell_5, \ell_3)) \\
\mathit{loc}(V, V') &= (\mathit{skip}(\mathtt{n}, \mathtt{ret}) \wedge \mathit{move}(\ell_2, \ell_3))
\end{aligned}
$$

Figure 1. An example program and its representation as a transition system. $\mathit{skip}(v_1, \ldots, v_k)$ abbreviates the conjunction $v_1' = v_1 \wedge \cdots \wedge v_k' = v_k$. $\mathit{move}(\ell, \ell')$ abbreviates the conjunction $\mathtt{pc} = \ell \wedge \mathtt{pc}' = \ell'$.

lieve that our method offers promising potential to boost the development (and deployment) of software verifiers in the same way the parser generators paved the way to modern approaches to compiler construction.

Our method focuses on automatic construction of verification tools that implement proof rules for reachability and termination properties in the form of Horn(-like) clauses, see e.g. [30]. Proof rules in such form are shown to be sufficiently expressive and practically adequate for dealing with a wide range of programming languages (including sequential, concurrent, and functional programs), temporal specifications and verification techniques, see e.g. [13, 18, 19, 37, 42, 43]. We present an efficient algorithm for solving Horn-like clauses by generalizing state-of-the-art software verification algorithms, in particular counterexample guided abstraction and refinement schemes. To demonstrate the feasibility of our approach in practice, we apply our solving algorithm in combination with appropriate constraint generation frontends to develop a collection of verifiers for temporal properties of programs with procedures, multi-threaded programs, and functional programs.

This paper makes the following contributions.

- Conceptually, we identify a formulation of proof rules for reachability and termination properties in the form of Horn-like clauses suitable for automation using state-of-the-art techniques for software verification.
- Technically, we provide an algorithm for solving Horn-like clauses that is based on a generalization of the state-of-the-art counterexample guided abstraction refinement schemes.
- Practically, we present an implementation of our approach and its evaluation on important proof rules for the verification of transition systems, programs with procedures, multi-threaded programs, and functional programs.

2. Illustration

We illustrate our verification approach using a simple example for which we prove a safety and a termination property.

2.1 Verifying programs with procedures

See Figure 1 for an example program implementing a sum computation. The procedure `sum` takes an argument `n` and returns the sum of the numbers between 1 and `n`, or 0 if its argument `n` is not positive. We assume that the program state is given by a valuation of the program variables $V = (\mathtt{n}, \mathtt{s}, \mathtt{ret}, \mathtt{pc})$. The variable `ret` models return value passing, while `pc` is a procedure-local program counter variable that keeps track of the current control location. The initial states of the program are given as an assertion $\mathit{init}(V)$, where ℓ_1 indicates the program line labeled 1. We use assertions over a tuple of variables V and its primed version V' to model program statements as binary relations over states. In our program, $\rho(V, V')$ represents intra-procedural statements. $\mathit{call}(V, V')$ and $\mathit{ret}(V, V')$ model parameter and return value passing, respectively. The last assertion $\mathit{loc}(V, V')$ ensures that the caller's local data variables are unchanged during the callee's execution, while the caller's program counter moves over the call site. The set of local variables that are kept unchanged by the assertion $\mathit{loc}(V, V')$ excludes `s` , which is assigned by the call statement.

Proving safety We prove that `sum` always returns a non-negative value and formalize this property by the assertion $\mathit{error}(V) = (\mathtt{pc} = \ell_5 \wedge \mathtt{ret} < 0)$. To prove the property, we rely on a summarization proof rule for programs with procedures [42]. This proof rule requires the construction of an auxiliary assertion $\mathit{Summ}(V, V')$ that represents a binary relation between entry states of a procedure and their successors on the same level of recursion. The following constraints over $\mathit{Summ}(V, V')$ guarantee that all necessary pairs of states are captured.

$$
\begin{aligned}
&\mathit{init}(V) \wedge V = V' \rightarrow \mathit{Summ}(V, V') \\
&\mathit{Summ}(V, V') \wedge \rho(V', V'') \rightarrow \mathit{Summ}(V, V'') \\
&\mathit{Summ}(V, V') \wedge \mathit{call}(V', V'') \wedge V'' = V''' \rightarrow \mathit{Summ}(V'', V''') \\
&\mathit{Summ}(V, V') \wedge \mathit{call}(V', V'') \wedge \mathit{Summ}(V'', V''') \wedge \\
&\qquad \mathit{ret}(V''', V'''') \wedge \mathit{loc}(V', V'''') \rightarrow \mathit{Summ}(V, V'''')
\end{aligned}
$$

We obtain a correctness proof if we can find an instance of Summ that satisfies the above constraints together with the implication $\mathit{Summ}(V, V') \wedge \mathit{error}(V') \rightarrow \mathit{false}$. The following disjunction is a solution that is computed by our proposed Horn solving algorithm.

$$
\begin{aligned}
\mathit{Summ}(V, V') = &(\mathtt{pc} = \ell_1 \wedge \mathtt{pc}' \in \{\ell_1, \ell_4\}) \vee \\
&(\mathtt{pc} = \ell_1 \wedge \mathtt{pc}' = \ell_2 \wedge \mathtt{n}' \geq 1) \vee \\
&(\mathtt{pc} = \ell_1 \wedge \mathtt{pc}' = \ell_3 \wedge \mathtt{n}' + \mathtt{s}' \geq 1) \vee \\
&(\mathtt{pc} = \ell_1 \wedge \mathtt{pc}' = \ell_5 \wedge \mathtt{ret}' \geq 0)
\end{aligned}
$$

Note that the last disjunct ensures the non-negativeness of the return value.

Proving termination For proving that `sum` terminates on every input, we require that a so-called recursion relation between entry states of the caller and its immediate callee is well-founded, i.e., it does not admit infinite chains. We obtain the recursion relation by relational composition of the summary with the parameter passing relation. Hence, for proving termination, we need to find an assertion $\mathit{Summ}(V, V')$ that satisfies the above implications ensuring the summarization property and the following well-foundedness condition.

$$\mathit{well\text{-}founded}(\mathit{Summ}(V, V') \wedge \mathit{call}(V', V''))$$

To prove termination, our previous solution for $Summ(V, V')$ requires a strengthening $\mathtt{n}' \leq \mathtt{n}$ that keeps track of changes applied to $\mathtt{n}$ and guarantees that $\mathtt{n}$ never increases. Our solving algorithm computes the following solution.

$$\begin{aligned} Summ(V, V') = {} & (\mathtt{pc} = \ell_1 \wedge \mathtt{pc}' = \ell_1 \wedge \mathtt{n} \geq \mathtt{n}') \vee \\ & (\mathtt{pc} = \ell_1 \wedge \mathtt{pc}' = \ell_2 \wedge \mathtt{n}' \geq 1 \wedge \mathtt{n} \geq \mathtt{n}') \vee \\ & (\mathtt{pc} = \ell_1 \wedge \mathtt{pc}' = \ell_3 \wedge \mathtt{n}' + \mathtt{s}' \geq 1) \vee \\ & (\mathtt{pc} = \ell_1 \wedge \mathtt{pc}' = \ell_4) \vee \\ & (\mathtt{pc} = \ell_1 \wedge \mathtt{pc}' = \ell_5 \wedge \mathtt{ret}' \geq 0) \end{aligned}$$

The composition of the strengthened solution with the parameter passing relation is

$$(move(\ell_1, \ell_1) \wedge \mathtt{n}' \geq 0 \wedge \mathtt{n}' \leq \mathtt{n} - 1) \ .$$

Its well-foundedness follows from the decrease of $\mathtt{n}$ at each step and its boundedness from below by $\mathtt{n}' \geq 0$.

As our example illustrates, we consider proof rules that can be represented as a set of implication constraints over formulas representing the program and "unknown" formulas, e.g. $Summ(V, V')$. To provide a basis for efficient solving algorithms, our implication constraints resemble Horn clauses as they may have at most one unknown formula in the consequent part of a clause. We refer to such implications as Horn-*like* clauses, since we allow arbitrary disjunctions among "known" formulas.

2.2 Verifying functional programs

We represent the program from Figure 1 as a functional program.

```
let rec sum n =
  if n > 0 then let s = sum (n-1) in s+n
  else 0
```

For functional programs, typing constraints can be used to track value flow through program expressions. They relate refinement types [14, 26] that represent assertions over values of expressions and values of identifiers in scope. For example, the type of the above function `sum` can be represented as follows.

$$\mathtt{sum} : (\mathtt{n} : \{\nu : \mathtt{int} \mid P_1(\nu)\} \rightarrow \{\nu : \mathtt{int} \mid P_2(\mathtt{n}, \nu)\})$$

Following recent work on refinement type inference, see e.g., [24, 48, 50], we embed the typing constraints into logical implications and obtain the following set of Horn-like clauses over $P_1(\nu)$ and $P_2(\mathtt{n}, \nu)$.

$$\begin{aligned} & \mathit{true} \rightarrow P_1(\nu) \\ & P_1(\mathtt{n}) \wedge \mathtt{n} > 0 \wedge \nu = \mathtt{n} - 1 \rightarrow P_1(\nu) \\ & P_1(\mathtt{n}) \wedge \mathtt{n} > 0 \wedge P_2(\mathtt{n} - 1, \nu) \wedge \nu' = \mathtt{n} + \nu \rightarrow P_2(\mathtt{n}, \nu') \\ & P_1(\mathtt{n}) \wedge \mathtt{n} \leq 0 \wedge \nu = 0 \rightarrow P_2(\mathtt{n}, \nu) \\ & P_2(\mathtt{n}, \nu) \rightarrow \nu \geq 0 \end{aligned}$$

The first clause encodes that there is no restriction on the inputs to `sum`. The second and third clauses represent data flow if the branching condition succeeds. First, there is data flow due to the parameter passing, which is represented by the second clause. Then, the result of the recursive call is represented using the assertion $P_2(\mathtt{n}-1, \nu)$. The sum of ν and $\mathtt{n}$ is the return value. The fourth clause encodes return value passing when the branching condition fails. The last clause represents the property that `sum` returns non-negative values. If there is a solution to the above clauses then `sum` satisfies the property.

Our solving algorithm computes solutions for $P_1(\nu)$ and $P_2(\mathtt{n}, \nu)$, which yields the following type for `sum`.

$$\mathtt{sum} : (\mathtt{n} : \{\nu : \mathtt{int} \mid \mathit{true}\} \rightarrow \{\nu : \mathtt{int} \mid \nu \geq 0\}) \ .$$

3. Preliminaries

In this section we introduce preliminary definitions.

We write $\lambda x \in X.e$ to represent a definition that assigns to each $x \in X$ the value obtained by evaluating e. Given a function f, let $dom(f)$ denote the domain of f, i.e., the set of values for which f is defined. A binary relation is well-founded if it does not admit infinite chains. We write $\mathit{well\text{-}founded}(\varphi(v, v'))$ if $\varphi(v, v')$ is a well-founded relation, i.e., there is no infinite sequence $s_1, s_2, \ldots$ such that $\varphi(s_i, s_{i+1})$ for all $i \geq 1$. Let ϵ denote the empty tuple. A relation $\varphi(v, v')$ is disjunctively well-founded if it is included in a finite union of well-founded relations, i.e., if there exist well-founded $\varphi_1(v, v'), \ldots, \varphi_n(v, v')$ such that

$$\varphi(v, v') \models_{\mathcal{T}} \varphi_1(v, v') \vee \cdots \vee \varphi_n(v, v').$$

For example, the relation $x \geq 0 \wedge x' \leq x - 1$ is well-founded, while the relation $x \geq 0 \wedge x' \leq x - 1 \vee y \leq 0 \wedge y' \geq y + 1$ is disjunctively well-founded.

Constraints Let $\mathcal{T}$ be a first-order theory in a given signature, $\mathcal{V}$ be a set of variables, and $\models_{\mathcal{T}}$ be the entailment relation with respect to $\mathcal{T}$. We write v to denote a non-empty tuple of variables, i.e., $v \in \mathcal{V}^+$. We refer to formulas in the given signature as constraints, and let $c(v)$ denote a constraint over the variables v. Let false denote an unsatisfiable constraint.

For example, let x, y, and z be variables. Then, $v = (x, y)$ and $w = (y, z)$ are tuples of variables. $x \leq 2$, $y \leq 1 \wedge x - y \leq 0$, and $f(x) + g(x, y) \leq 3 \vee z \leq 0$ are example constraints in the theory $\mathcal{T}$ of linear inequalities over rationals/reals and uninterpreted functions, where f and g are uninterpreted function symbols. The entailment $y \leq 1 \wedge x - y \leq 0 \models_{\mathcal{T}} x \leq 2$ is valid.

Queries and dwf-predicates We assume a set of uninterpreted predicate symbols $\mathcal{Q}$ that we refer to as query symbols. The arity of a query symbol is encoded in its name. We write q to denote a query symbol. Given q of a non-zero arity n and a tuple of variables v of length n, we define $q(v)$ to be a query. Furthermore, we introduce an interpreted predicate symbol dwf of arity one (dwf stands for disjunctive well-foundedness). Given a query $q(v, v')$ over tuples of variables with equal length, we refer to $dwf(q(v, v'))$ as a dwf-predicate.

For example, let $\mathcal{Q} = \{r, s\}$ be query symbols of arity one and two, respectively. Then, $r(x)$ and $s(x, y)$ are queries, and $dwf(s(x, y))$ is a dwf-predicate.

Horn-like clauses Let $h(v)$ range over queries and constraints with variables in v. We define a Horn-like clause to be either an implication

$$c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$$

or a unit clause

$$dwf(q(v, v'))$$

which consists of a dwf-predicate. The left-hand side of the implication is called the body and the right-hand side is called the head. We use cl to denote a Horn-like clause.

The following set of clauses $\mathcal{C}$ illustrates our definition of Horn-like clauses.

$$\begin{aligned} \mathcal{C} = \{ & x \leq y \wedge y \leq -1 \rightarrow r(x), y = x + 1 \wedge r(x) \rightarrow s(x, y), \\ & r(x) \rightarrow x \leq 0, dwf(s(x, y))\} \end{aligned}$$

To support efficient verification, our Horn-like clauses slightly deviate from the standard notion of Horn clauses since constraints occurring in our clauses can contain disjunctions and conjunctions. For example, we admit clauses such as

$$(x \leq 0 \vee y \leq 0) \wedge s(x, y) \rightarrow s(x, y)$$

and

$$s(x, y) \rightarrow (x \leq 0 \vee y \leq 0).$$

While our presentation of the proposed method does not rely on the Boolean structure of constraints occurring in clauses, it is useful in practice to allow disjunction in constraints in order to keep the number of clauses small. (Note that the above two clauses can be translated into logically equivalent Horn clauses $x \leq 0 \wedge s(x, y) \rightarrow s(x, y)$, $y \leq 0 \wedge s(x, y) \rightarrow s(x, y)$ and $s(x, y) \wedge \neg(x \leq 0) \rightarrow y \leq 0$.) In contrast, we rely on the fact that there is at most one non-negated query in a clause.

Clauses in normal form Before formalizing the semantics of the clauses, we introduce assumptions on the syntax of Horn-like clauses that significantly simplify the presentation of the semantics without introducing any proper restrictions.

First, we assume that for each query symbol q there is a fixed tuple v of variables with the corresponding length and that each query with the symbol q is of the form $q(v)$. That is, each query has an *a priori* defined tuple of variables. Furthermore, we assume that all variables in v are pairwise distinct, i.e., for the tuple of variables $v = (x_1, \ldots, x_n)$ we have x_i and x_j are different variables for all $1 \leq i \neq j \leq n$.

Second, we assume that in each clause a query symbol can occur at most once. Formally, for each clause $c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$ we assume that q_i is different from q_j for all $1 \leq i \neq j \leq n$, and if the head $h(v)$ is a query $q(v)$ then q is different from each q_i for all $1 \leq i \leq n$.

The first assumption can be established by assigning tuples of variables to query symbols, and then translating queries into the desired form by adding corresponding equality constraints into the constraint of a clause. For example, for the query symbols r and s we assign variables $v_r = (x_r)$ and $v_s = (x_s, y_s)$, respectively. Then, a clause $x + y \leq 0 \wedge r(x) \wedge s(x, x) \rightarrow y \leq 0$ violates our first assumption but can be transformed to

$$x = x_r \wedge x = x_s \wedge x = y_s \wedge \\ x + y \leq 0 \wedge r(x_r) \wedge s(x_s, y_s) \rightarrow y \leq 0.$$

The first conjunct corresponds to the translation of the query $r(x)$, while the second and third conjuncts correspond to $s(x, x)$.

The second assumption can be established by introducing auxiliary queries and clauses each time there is a clause with multiple occurrences of some query. The violating clause is transformed by replacing the violating query occurrences in the clause body by the auxiliary queries.

For example, a clause $r(x_r) \wedge r(x_r) \rightarrow r(x_r)$ violates the second assumption due to the triple occurrence of $r(x_r)$. Hence, we introduce two auxiliary query symbols r_1 and r_2 of arity one together with the corresponding tuples of variables $v_{r_1} = (x_{r_1})$ and $v_{r_2} = (x_{r_2})$, respectively. Then, we express the relation between $r(x_r)$ and the introduced queries using the following auxiliary clauses: $x_r = x_{r_1} \wedge r(x_r) \rightarrow r_1(x_{r_1})$ and $x_r = x_{r_2} \wedge r(x_r) \rightarrow r_2(x_{r_2})$. Finally, we translate the original clause $r(x_r) \wedge r(x_r) \rightarrow r(x_r)$ to

$$x_r = x_{r_1} \wedge x_r = x_{r_2} \wedge r_1(x_{r_1}) \wedge r_2(x_{r_2}) \rightarrow r(x_r).$$

We refer to a set of Horn-like clauses that satisfies the above two conditions as clauses in normal form. In the rest of the paper, we assume that the clauses are Horn-like and in normal form.

For $\mathcal{C}$ defined above we obtain the following normal form $\mathcal{C}_{NF}$.

$$\{x = x_r \wedge x \leq y \wedge y \leq -1 \rightarrow r(x_r), \\ x = x_r \wedge x = x_s \wedge y = y_s \wedge y = x + 1 \wedge r(x_r) \rightarrow s(x_s, y_s), \\ x = x_r \wedge r(x_r) \rightarrow x \leq 0, \mathit{dwf}(s(x_s, y_s))\}$$

Semantics of Horn-like clauses A set of clauses can be seen as an assertion over the queries that occur in the clauses.

We consider a function Σ that maps each query $q(v)$ occurring in a given set of clauses into a constraint over v. Such a function is called a solution if the following two conditions hold. First, for each clause $c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$ from the given set we require:

$$c(v_0) \wedge \Sigma(q_1) \wedge \cdots \wedge \Sigma(q_n) \models_{\mathcal{T}} \begin{cases} \Sigma(q) & \text{if } h(v) \text{ is } q(v), \\ c_h(v) & \text{if } h(v) \text{ is } c_h(v). \end{cases}$$

Second, for each clause $\mathit{dwf}(q(v, v'))$ in the input set we require that the relation $\Sigma(q)$ is disjunctively well-founded. Let $\models_{\mathcal{Q}}$ be the corresponding satisfaction relation, i.e., $\Sigma \models_{\mathcal{Q}} \mathcal{C}$ if Σ is a solution for $\mathcal{C}$.

For example, the previously defined set of clauses $\mathcal{C}_{NF}$ has a solution Σ such that

$$\Sigma(r) = x_r \leq -1, \\ \Sigma(s) = x_s \leq 0 \wedge y_s \geq x_s + 1.$$

To check $\Sigma \models_{\mathcal{Q}} \mathcal{C}_{NF}$ we consider the validity of the entailments

$$x = x_r \wedge x \leq y \wedge y \leq -1 \models_{\mathcal{T}} x_r \leq -1, \\ x = x_r \wedge x = x_s \wedge y = y_s \wedge y = x + 1 \wedge x_r \leq -1 \\ \models_{\mathcal{T}} x_s \leq 0 \wedge y_s \geq x_s + 1, \\ x = x_r \wedge x_r \leq -1 \models_{\mathcal{T}} x \leq 0,$$

and the fact that $\Sigma(s)$ is a (disjunctively) well-founded relation.

Dependency and recursion-free clauses For a clause cl such that

$$c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$$

we define $\mathit{depends}(cl)$ to be the set of query symbols that appear in the body of cl, i.e., $\mathit{depends}(cl) = \{q_1, \ldots, q_n\}$. A set of clauses defines a binary dependency relation on query symbols. Each clause cl that has a query $q(v)$ (rather than a constraint) in its head contributes the set of pairs $\{(q_i, q) \mid q_i \in \mathit{depends}(cl)\}$ to the dependency relation. We say that a set of clauses is recursion-free if the corresponding dependency relation is well-founded.

For example, the second clause in $\mathcal{C}$ depends on the set of query symbols $\{r\}$, and the entire set of clauses $\mathcal{C}$ defines the dependency relation $\{(r, s)\}$. This dependency relation is well-founded, hence $\mathcal{C}$ is recursion-free.

Solving recursion-free clauses We assume an algorithm for solving recursion-free clauses. This algorithm takes as input recursion-free clauses in the theory $\mathcal{T}$ and computes a solution Σ when it exists. There already exist such algorithms for the theory of linear arithmetic (see [19]) and linear arithmetic with uninterpreted functions (see [17]), which are based on extensions of interpolation algorithms [32, 46] to tree-like structures.

4. Algorithm HSF

In this section we present our algorithm HSF for finding solutions to recursive Horn-like clauses.

Let $\mathcal{C}$ be a finite set of clauses that is given as input to HSF. We partition $\mathcal{C}$ into inference clauses $\mathcal{I}$ and property clauses $\mathcal{P}$. Inference clauses contain queries in their heads, and property clauses contain the rest, i.e.,

$$\mathcal{I} = \{cl \in \mathcal{C} \mid cl = (\ldots \rightarrow q(v))\}, \\ \mathcal{P} = \mathcal{C} \setminus \mathcal{I}.$$

Inference clauses impose a relationship between queries, while property clauses impose absolute assertions on queries.

HSF finds a solution by following an iterative, abstraction-based approach that relies on (spurious) counterexample derivations to refine the abstraction in case of imprecision. This approach

is a generalization of the counterexample-guided abstraction refinement schemes for proving reachability and termination properties of software. Our generalization deals with Horn-like clauses (instead of transition systems/programs with procedures). Our approach inherits the advantages and disadvantages of the existing counterexample-guided abstraction refinement schemes: a sufficiently precise yet not overly detailed abstraction can be discovered automatically, however the abstraction discovery procedure may not terminate. In practice, the non-terminating behavior is sufficiently seldom.

The iteration proceeds in three main steps.

1. We find a solution for the inference clauses $\mathcal{I}$. At this step we perform logical inference and rely on abstraction to ensure termination in the presence of recursion and to ensure efficiency in the presence of large sets of clauses.
2. We check whether the computed solution satisfies the property clauses $\mathcal{P}$. If some property clause, say cl, is not satisfied then we proceed with the analysis of the inference tree computed in the first step. Otherwise, if all property clauses are satisfied, we return the solution.
3. We check whether the logical inference performed in the first step in the setting without any abstraction yields a solution that still violates the property clause cl. If the violation is present then we return the inference tree as a counterexample derivation. Otherwise we use the obtained solution to refine the abstraction function and go back to the first step.

The first step is implemented using a procedure INFERABST that applies ADDINFERRED to perform the necessary bookkeeping of the inference tree construction. MAKECEX extracts a relevant subtree of the inference tree in case a solution computed by INFERABST violates some property clause. We rely on existing procedures for the analysis of the obtained subtree. ADDPREDS converts solutions obtained by the successful subtree analysis into a refinement of the abstraction function. The procedure HSF puts the steps together in a loop (Figure 4).

Next we present the procedures that implement the three steps.

(Predicate) abstraction We use predicate abstraction as an approximation technique employed by HSF.

Let α be a function that takes as input a constraint $\varphi(v)$ together with a finite set $\{c_1(v), \ldots, c_n(v)\}$ of predicates, i.e., atomic constraints, over v. The output is an over-approximation of $\varphi(v)$ that is constructed from the given predicates using Boolean operators. We use the following definition (which is called Cartesian abstraction in the literature [5]).

$$\alpha(\varphi(v), \{c_1(v), \ldots, c_n(v)\}) \;=\; \bigwedge\{c_i(v) \mid i \in 1..n \wedge \varphi(v) \models_\tau c_i(v)\}$$

For example, given the constraint $x \leq y \wedge y \leq z \wedge z \leq 0$ and the predicates $\{x \leq z, x \geq 0, x \leq 0\}$, the predicate abstraction function returns the conjunction $x \leq z \wedge x \leq 0$.

We rely on two properties of the abstraction function: over-approximation and monotonicity. That is, for each pair of constraints $\varphi(v)$, $\psi(v)$ and each set of predicates $Preds$ we have i) $\varphi(v)$ entails $\alpha(\varphi(v), Preds)$ and ii) if $\varphi(v)$ entails $\psi(v)$ then $\alpha(\varphi(v), Preds)$ entails $\alpha(\psi(v), Preds)$. The over-approximation will guarantee that combining logical inference with abstraction will yield solutions to inference clauses and the monotonicity will guarantee that such solutions can be computed using fixpoint iteration techniques.

Inference and abstraction Before presenting the procedure INFERABST, which performs logical inference, abstraction, and bookkeeping, we first present a characterization of what it computes in the form of inference rules.

$$\text{RINIT}\ \frac{c(v_0) \to q(v) \in \mathcal{I}}{\alpha(c(v_0), Preds(q)) \in Inferred(q)}$$

$$\text{RSTEP}\ \frac{\varphi_1(v_1) \in Inferred(q_1) \quad \ldots \quad \varphi_n(v_n) \in Inferred(q_n) \qquad c(v_0) \wedge \bigwedge_{i=1}^{n} q_i(v_i) \to q(v) \in \mathcal{I}}{\alpha(c(v_0) \wedge \bigwedge_{i=1}^{n} \varphi_i(v_i), Preds(q)) \in Inferred(q)}$$

Figure 2. Abstract inference rules for a given set of clauses $\mathcal{I}$ and a predicate abstraction function with the set of predicates $Preds(q)$ for each query symbol q.

See Figure 2. The presented rules RINIT and RSTEP define a relation between a possibly empty sequence of constraints in the premise and a constraint in the consequence such that the relation satisfies some inference clause from the input set $\mathcal{I}$.

We keep track of the inferred constraints in the set $Inferred$ that we partition according to the query symbols of these constraints. The inference process applies the rules as long as the derived constraints are not subsumed by the previously derived ones. A constraint $\varphi(v)$ derived by applying a clause with the query $q(v)$ in its head is subsumed if there is a constraint $\psi(v)$ in $Inferred(q)$ such that $\varphi(v)$ entails $\psi(v)$. (This subsumption definition is called local entailment in the literature.)

We observe that the inference process terminates since the range of the abstraction function is finite, i.e., there are only finitely many different constraints that can be added to $Inferred$, and the abstraction function is monotonic. Furthermore, the resulting constraints yield a solution for the inference clauses. Formally, we define

$$\Sigma \;=\; \lambda q \in dom(Inferred).\, \bigvee Inferred(q)$$

and obtain $\Sigma \models_{\mathcal{Q}} \mathcal{I}$.

***Procedure* INFERABST** We turn the inference rules RINIT and RSTEP into a worklist-based iteration procedure INFERABST that also keeps track between inferred constraints and corresponding clauses. See Figure 3.

INFERABST takes as input the inference clauses $\mathcal{I}$ and a function $Preds$ that assigns to each query symbol q (and the corresponding tuple of variables v) a finite set of predicates over v. The output of INFERABST consists of the inferred constraints $Inferred$ and a function $Parent$ that represent the bookkeeping results. $Parent$ assigns to each inferred constraint $\varphi(v)$ a sequence of constraints and a clause that were used in the rule application that produced $\varphi(v)$. INFERABST maintains a worklist WL containing inference clauses that may infer new constraints.

The inference starts with applying all clauses that do not depend on any queries, and are hence applicable when no constraints are yet inferred. Each clause application follows the rule RINIT and applies ADDINFERRED to process the application result. Then, INFERABST iteratively applies the clauses from the worklist until no more non-subsumed constraints can be computed, i.e., until the worklist becomes empty. At every iteration step, INFERABST takes a clause from the worklist and exhaustively applies the clause following the rule RSTEP.

Example For the set of clauses $\mathcal{C}_{NF}$ defined in the previous section, we consider the predicates $Preds(r) = \{x_r \leq 0\}$ and $Preds(s) = \{x_s \leq y_s\}$. For brevity, we denote the four clauses from $\mathcal{C}_{NF}$ as cl_1 , cl_2 , cl_3, and cl_4 , respectively. Our algorithm initially processes the clauses that do not depend on any queries,

```
    function INFERABST
    input
      I – inference clauses
      Preds – predicate table
    output
      Inferred – inferred constraints
      Parent – parent function for inferred constraints
    vars
      WL – worklist with clauses
      procedure ADDINFERRED
      input
        φ(v) – abstract relation
        φ1(v1), ..., φn(vn) – parent abstract relations
        cl = (... → q(v)) – parent clause with head q(v)
      begin
1       if ¬(∃ψ(v) ∈ Inferred(q) : φ(v) ⊨T ψ(v)) then
2         Inferred(q) := {φ(v)} ∪ Inferred(q)
3         Parent(φ(v)) := ((φ1(v1), ..., φn(vn)), cl)
4         WL := {cl' ∈ I | q ∈ depends(cl')} ∪ WL
      end
    begin
5     Inferred := λq ∈ Q.∅
6     Parent := ∅
7     WL := ∅
8     for each c(v0) → q(v) ∈ I do
9       φ(v) := α(c(v0), Preds(q))
10      ADDINFERRED(φ(v), ε, c(v0) → q(v))
11    while WL ≠ ∅ do
12      c(v0) ∧ q1(v1) ∧ ··· ∧ qn(vn) → q(v) := take from WL
13      for each i ∈ 1..n and φi(vi) ∈ Inferred(qi) do
14        φ(v) := α(c(v0) ∧ φ1(v1) ∧ ··· ∧ φn(vn), Preds(q))
15        ADDINFERRED(φ(v), (φ1(v1), ..., φn(vn)),
                      c(v0) ∧ q1(v1) ∧ ··· ∧ qn(vn) → q(v))
16    return (Inferred, Parent)
    end
```

Figure 3. Abstract inference algorithm.

i.e., cl_1 (see lines 8–10 from Figure 3). The computation of a new constraint at line 9 proceeds as follows: $\varphi(x_r) = \alpha(x = x_r \wedge x \leq y \wedge y \leq -1, \{x_r \leq 0\}) = (x_r \leq 0)$. □

***Procedure* ADDINFERRED** The procedure ADDINFERRED is shown in Figure 3. Since ADDINFERRED is defined within INFERABST, *Inferred*, *Parent*, and WL are in scope of ADDINFERRED.

The input to ADDINFERRED is an inferred constraint $\varphi(v)$ that was computed by applying a clause cl with the head $q(v)$ on the possibly empty sequence of constraints $\varphi_1(v_1), \ldots, \varphi_n(v_n)$, i.e., n may be equal to zero. First, ADDINFERRED checks if $\varphi(v)$ is subsumed by already inferred constraints. If no subsumption takes place then we add it to the set of inferred constraints *Inferred*, extend *Parent* with a corresponding bookkeeping record, and add inference clauses that depend on q to the worklist.

Example (cont.) After computing the constraint $\varphi(x_r) = x_r \leq 0$, the inference algorithm calls ADDINFERRED$(x_r \leq 0, \epsilon, cl_1)$ (see line 10 of Figure 3) and records the information about the newly inferred constraint as follows:

$$\begin{aligned} \mathit{Inferred}(r) &= \{x_r \leq 0\} \,, \\ \mathit{Parent}(x_r \leq 0) &= (\epsilon, cl_1) \,, \\ WL &= \{cl_2\} \,. \end{aligned}$$

After adding the clause cl_2 to the worklist, it will be processed in the loop at lines 11–15 and a call ADDINFERRED$(x_s \leq y_s, (x_r \leq 0), cl_2)$ leads to a second inferred constraint as follows:

$$\begin{aligned} \mathit{Inferred}(s) &= \{x_s \leq y_s\} \,, \\ \mathit{Parent}(x_s \leq y_s) &= ((x_r \leq 0), cl_2) \,, \\ WL &= \emptyset \,. \end{aligned}$$

For the given sets of predicates, i.e., $\mathit{Preds}(r) = \{x_r \leq 0\}$ and $\mathit{Preds}(s) = \{x_s \leq y_s\}$, the computation of inferred constraints finishes here since the worklist is empty. □

***Procedure* HSF** The main procedure of our algorithm is HSF. It is shown in Figure 4. HSF takes as input a finite set of Horn-like clauses $\mathcal{C} = \mathcal{I} \uplus \mathcal{P}$ and iteratively computes a solution for the inference clauses $\mathcal{I}$ that also satisfies the property clauses $\mathcal{P}$.

HSF computes solution candidates from the constraints in *Inferred* that are inferred using INFERABST. The obtained candidate is guaranteed to satisfy $\mathcal{I}$, while satisfaction of $\mathcal{P}$ requires finding a sufficiently precise abstraction function. The abstraction function α is determined by a set of predicates *Preds* that is partitioned between query symbols.

A sufficiently precise abstraction function is computed by HSF iteratively by adding predicates to *Preds*, which is empty initially. For the given *Preds*, we first use INFERABST to compute inferred constraints *Inferred* and *Parent*. Then we check whether the solution for $\mathcal{I}$ defined by *Inferred* also satisfies $\mathcal{P}$. We distinguish between violation of a clause that has a constraint in its head from the violation of a clause consisting of a dwf-predicate.

If a clause $cl = c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow c_h(v)$ is not satisfied by some sequence of inferred constraints, say $\varphi_1(v_1) \in \mathit{Inferred}(q_1), \ldots, \varphi_n(v_n) \in \mathit{Inferred}(q_n)$, then we check if repeating the same inference steps without applying the abstraction function computes a sequence of constraints that satisfies cl.

We reconstruct the inference steps that produced $\varphi_1(v_1), \ldots, \varphi_n(v_n)$ by using the procedure MAKECEX. The output of MAKECEX is a set of recursion-free clauses $\mathcal{X}$ whose queries correspond to the constraints that were involved, as recorded by *Parent*, in the computation of $\varphi_1(v_1), \ldots, \varphi_n(v_n)$. We record the correspondence using the function *Sym* that assigns query symbols of the involved constraints to the fresh query symbols that represent constraints derived without abstraction. We also include into $\mathcal{X}$ a clause imposing an assertion on the queries that correspond to $\varphi_1(v_1), \ldots, \varphi_n(v_n)$.

Now, we solve $\mathcal{X}$ using an existing tool for solving recursion-free clauses [17, 19]. If a solution exists then we use it to extract additional predicates for *Preds*. The function *Sym* translates query symbols from the domain of the solution into the query symbols in our clauses $\mathcal{C}$. In this case, after refining the abstraction we continue with another attempt to find a solution for $\mathcal{C}$. If a solution does not exist, we return the clauses $\mathcal{X}$ as a witness of the property violation.

Example (cont.) The set of clauses $\mathcal{C}_{NF}$ contains the property clause cl_3, i.e., $x = x_r \wedge r(x_r) \rightarrow x \leq 0$, which is satisfied by the candidate solution: $\mathit{Inferred}(r) = (x_r \leq 0)$. □

If a clause $dwf(q(v, v'))$ is not satisfied by some of the inferred constraints, say $\varphi(v, v') \in \mathit{Inferred}(q)$, then we proceed in a similar way as in the above case. First, we use MAKECEX to construct a set of recursion-free clauses $\mathcal{X}$ that reconstructs the inference steps leading to $\varphi(v, v')$. Then, we rely on an existing solver for recursion-free clauses to find a solution for $\mathcal{X}$ that assigns

function HSF
input
$\mathcal{I} \uplus \mathcal{P}$ – Horn-like inference and property clauses
vars
$Preds$ – predicate table
Sym – definition of quoted query symbols
function MAKECEX
input
$\varphi(v)$ – inferred constraint
$Parent$ – parent function
output
$\mathcal{X}$ – set of recursion-free clauses with root $\varphi(v)$
begin
1 $((\varphi_1(v_1), \ldots, \varphi_n(v_n)),$
$c(v_0) \wedge \bigwedge_{i=1}^n q_i(v_i) \rightarrow h(v)) := Parent(\varphi(v))$
2 $Sym := \{\text{“}\varphi_1(v_1)\text{”} \mapsto q_1, \ldots, \text{“}\varphi_n(v_n)\text{”} \mapsto q_n\} \cup Sym$
3 **return** $\{c(v_0) \wedge \bigwedge_{i=1}^n \text{“}\varphi_i(v_i)\text{”}(v_i) \rightarrow \text{“}\varphi(v)\text{”}(v)\} \cup$
4 $\bigcup_{i=1}^n \text{MAKECEX}(\varphi_i(v_i), Parent)$
end
procedure ADDPREDS
input
Σ – solution function
begin
5 **for** each $\text{“}\varphi(v)\text{”} \in dom(\Sigma)$ **do**
6 $q := Sym(\text{“}\varphi(v)\text{”})$
7 $Preds(q) := \Sigma(\text{“}\varphi(v)\text{”}) \cup Preds(q)$
end
begin
8 $Preds := \lambda q \in \mathcal{Q}.\emptyset$
9 **repeat**
10 $(Inferred, Parent) := \text{INFERABST}(\mathcal{I}, Preds)$
11 **if** exist $c(v_0) \wedge \bigwedge_{i=1}^n q_i(v_i) \rightarrow c_h(v) \in \mathcal{P}$ and
$\varphi_i(v_i) \in Inferred(q_i)$ for each $i \in 1..n$ such that
12 $c(v_0) \wedge \bigwedge_{i=1}^n \varphi_i(v_i) \not\models_\mathcal{T} c_h(v)$ **then**
13 $Sym := \{\text{“}\varphi_1(v_1)\text{”} \mapsto q_1, \ldots, \text{“}\varphi_n(v_n)\text{”} \mapsto q_n\}$
14 $\mathcal{X} := \{c(v_0) \wedge \bigwedge_{i=1}^n \text{“}\varphi_i(v_i)\text{”}(v_i) \rightarrow c_h(v)\} \cup$
15 $\bigcup_{i=1}^n \text{MAKECEX}(\varphi_i(v_i), Parent)$
16 **if** exists Σ such that $\Sigma \models_\mathcal{Q} \mathcal{X}$ **then**
17 ADDPREDS(Σ)
18 **else**
19 **return** “error derivation $\mathcal{X}$”
20 **else if** exist $dwf(q(v, v')) \in \mathcal{P}$ and $\varphi(v, v') \in Inferred(q)$
such that $\neg well\text{-}founded(\varphi(v, v'))$ **then**
21 $Sym := \{\text{“}\varphi(v, v')\text{”} \mapsto q\}$
22 $\mathcal{X} := \text{MAKECEX}(\varphi(v, v'), Parent)$
23 **if** exists Σ such that $\Sigma \models_\mathcal{Q} \mathcal{X}$ and
$well\text{-}founded(\Sigma(\text{“}\varphi(v, v')\text{”}))$ **then**
24 ADDPREDS(Σ)
25 **else**
26 **return** “error derivation $\mathcal{X}$”
27 **else**
28 **return** “solution $\lambda q \in dom(Inferred).\bigvee Inferred(q)$”
end.

Figure 4. Abstract inference, checking, and refinement.

a well-founded relation to $q(v, v')$ [39]. If INFERABST infers a set of constraints $Inferred$ that defines a solution for $\mathcal{P}$ then we return this solution.

Example (cont.) The set of clauses $\mathcal{C}_{NF}$ contains a second property clause, i.e., $dwf(s(x_s, y_s))$. The conditions at line 20 of Figure 4 are satisfied for the inferred constraint $x_s \leq y_s$: $dwf(s(x_s, y_s)) \in \mathcal{C}_{NF}$, $x_s \leq y_s \in Inferred(s)$ and $\neg well\text{-}founded(x_s \leq y_s)$. In this case, the inferred constraint does not correspond to a well-founded relation. □

***Procedure* MAKECEX** The procedure MAKECEX is shown in Figure 4. Its scope contains Sym and $Preds$ from HSF. MAKECEX takes as input an inferred constraint $\varphi(v)$ and bookkeeping records $Parent$ such that $\varphi(v) \in dom(Parent)$.

Then, MAKECEX creates a clause that records the fact that $\varphi(v)$ was derived using the constraints and the clause in $Parent(\varphi(v))$. Let $Parent(\varphi(v))$ be the pair of $\varphi_1(v_1), \ldots, \varphi_n(v_n)$ and $c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$. This dependency is modeled by introducing auxiliary queries $\text{“}\varphi_1(v_1)\text{”}(v_1), \ldots, \text{“}\varphi_n(v_n)\text{”}(v_n)$. These auxiliary queries stay in one-to-one correspondence with the constraints, i.e., $\text{“}\varphi_i(v_i)\text{”}(v_i)$ corresponds to q_i for all $1 \leq i \leq n$. This correspondence is established by applying a bijective quotation function $\text{“}\cdot\text{”}$ that translates a constraint into a query symbol. Finally, we recursively apply MAKECEX on the constraints that produced $\varphi(v)$ and return all constructed clauses.

Example (cont.) For the inferred constraint $s = x_s \leq y_s$, the procedure $\text{MAKECEX}(x_s \leq y_s, Parent)$ is invoked at line 22 from Figure 4. As a result, a function Sym is constructed to keep track of the quoted query symbols: $\{\text{“}x_s \leq y_s\text{”} \mapsto s, \text{“}x_r \leq 0\text{”} \mapsto r\}$. Finally, the procedure MAKECEX returns the following set of recursion-free clauses:

$$\{x = x_r \wedge x = x_s \wedge y = y_s \wedge y = x + 1 \wedge \text{“}x_r \leq 0\text{”}(x_r) \rightarrow \text{“}x_s \leq y_s\text{”}(x_s, y_s),$$
$$x = x_r \wedge x \leq y \wedge y \leq -1 \rightarrow \text{“}x_r \leq 0\text{”}(x_r)\}$$

□

***Procedure* ADDPREDS** The procedure ADDPREDS is shown in Figure 4. It has Sym in scope and takes as input a solution function whose domain consists of quoted constraints. ADDPREDS uses Sym to translate quoted constraints into original query symbols and add the solution constraints into the corresponding partitions of $Preds$.

Example (cont.) Let us assume that the following solution Σ is returned at line 23 of Figure 4: $\{\text{“}x_r \leq 0\text{”}(x_r) \mapsto x_r \leq 0, \text{“}x_s \leq y_s\text{”}(x_s, y_s) \mapsto x_s < y_s\}$. The solution constraints are partitioned as follows: $Preds(r) = \{x_r \leq 0\}$ and $Preds(s) = \{x_s \leq y_s, x_s < y_s\}$. Given these predicates, the next iteration of the abstract inference algorithm computes abstract constraints precise enough to satisfy both property clauses cl_3 and cl_4 from our example.

$$Inferred(r) = \{x_r \leq 0\}$$
$$Inferred(s) = \{x_s \leq y_s \wedge x_s < y_s\}$$

□

Correctness Upon termination, the algorithm HSF computes a solution for the input clauses. The soundness of the approach is guaranteed by the fact that the abstraction function is over-approximating. Our abstraction refinement method guarantees that a set of counterexample clauses $\mathcal{X}$ is never analyzed twice, i.e., our refinement method satisfies the progress of refinement property. Such soundness and progress of refinement properties are standard for counterexample guided abstraction refinement schemes.

For assertions $\{T_f$ over V_f and $V_f' \mid f \in Procs(P)\}$,

$$
\begin{array}{lll}
\text{CP1} : init(V_{main}) \wedge V_{main} = V'_{main} & \rightarrow T_{main}(V_{main}, V'_{main}) & \\
\text{CP2} : T_f(V_f, V_f') \wedge \rho_f(V_f', V_f'') & \rightarrow T_f(V_f, V_f'') & \\
\text{CP3} : T_f(V_f, V_f') \wedge call_{f,g}(V_f', V_g'') \wedge V_g'' = V_g''' & \rightarrow T_g(V_g'', V_g''') & f, g \in Procs(P) \text{ such that } f \text{ calls } g \\
\text{CP4} : T_f(V_f, V_f') \wedge call_{f,g}(V_f', V_g'') \wedge T_g(V_g'', V_g''') \wedge & & \\
\quad ret_{f,g}(V_g''', V_f'''') \wedge loc_f(V_f', V_f'''') & \rightarrow T_f(V_f, V_f'''') & f, g \in Procs(P) \text{ such that } f \text{ calls } g \\
\text{CP5} : T_f(V_f, V_f') \wedge error(V_f') & \rightarrow false &
\end{array}
$$

program P is safe

Figure 5. Summarization rule for programs with procedures.

Alternative solving methods In this paper we use predicate abstraction and refinement to solve Horn-like clauses. Predicate abstraction allows us to formulate a practical algorithm, yet the presented formulation is sufficiently general such that a different approximation techniques can be employed instead without any significant changes to HSF. For example, abstract domains based on widening can be used to compute a solution to inference clauses, by choosing the corresponding abstraction function for INFER-ABST. Alternatively, we could use model checking techniques, e.g., bounded model checking, to explore the clauses up to a finite bound. Usually, predicates are atomic constraints; however recent approaches to predicate abstraction show that predicates containing Boolean operators can be useful [7].

5. Proof rules as Horn-like clauses

In this section, we show a collection of proof rules that can be automated using our verification approach.

5.1 Transition systems

We consider a transition system with variables V, a set of initial states $init(V)$, a transition relation $\rho(V, V')$, and a standard semantics.

Safety Let $error(V)$ represent a set of error states. To verify safety the transition system, we use an invariance proof rule with conditions over a query $R(V)$ that characterizes reachable states as follows.

For assertion R over V ,

$$
\begin{array}{lll}
\text{CR1} : & init(V) & \rightarrow R(V) \\
\text{CR2} : & R(V) \wedge \rho(V, V') & \rightarrow R(V') \\
\text{CR3} : & R(V) \wedge error(V) & \rightarrow false
\end{array}
$$

transition system P is safe

Condition CR1 requires that all initial states are present in $R(V)$. Condition CR2 states that a program transition starting from a state in $R(V)$ ends in a state that is in $R(V')$. Finally, CR3 states that the intersection of reachable and error states is empty. Our algorithm HSF finds a solution for the query $R(V)$.

Termination To reason about termination properties of transition systems, we use a proof rule based on transition invariants [37]. The assertion T represents a transition invariant, which is a superset of the transitive closure of the transition relation ρ .

For assertion R over V that satisfies CR1 and CR2
and assertion T over V and V' ,

$$
\begin{array}{lll}
\text{CT1} : & R(V) \wedge \rho(V, V') & \rightarrow T(V, V') \\
\text{CT2} : & T(V, V') \wedge \rho(V', V'') & \rightarrow T(V, V'') \\
\text{CT3} : & dwf(T(V, V')) &
\end{array}
$$

transition system P terminates

We restrict the assertion $T(V, V')$ to states that are reachable using CT1, CT2 and two clauses from the invariance proof rule, CR1 and CR2 . The last clause, CT3, uses a predicate symbol dwf of arity one that requires a disjunctive well-founded argument $T(V, V')$. The existence of a (disjunctive well-founded) transition invariant guarantees program termination, cf. [37].

5.2 Programs with procedures

We consider programs with a set of recursive procedures $Procs(P)$. Let $main \in Procs(P)$ be the procedure that starts the program execution. For each $f \in Procs(P)$, let V_f the set of variables that are in scope, $\rho_f(V_f, V_f')$ be the intra-procedural transition relation. If f calls a procedure $g \in Procs(P)$, then let $call_{f,g}(V_f, V_g')$ and $ret_{f,g}(V_f, V_g')$ be parameter and return value passing relations, respectively. The relation $loc_f(V_f, V_f')$ states which local variables of f are not modified during a call to g.

Safety We prove safety properties of procedural programs using a rule based on context-free language reachability [42]. See Figure 5 for the proof rule that consists of constraints over assertions T_f , one for each program procedure. A query $T_f(V_f, V_f')$ represents a summary of the procedure f, which is a binary relation between entry states of f and their successors on the same level of recursion. For simplicity, our formulation does not adopt the common distinction between path edges and summaries, see, e.g., [42].

The first condition CP1 of the proof rule requires that the initial states constraint implies the query corresponding to the entry procedure $main$. The condition CP2 extends a query $T_f(V_f, V_f')$ with a transition relation from the same procedure. The third and fourth conditions handle procedure calls. In CP3, given a query $T_f(V_f, V_f')$ of the caller and the calling context passed from the variables V_f' to V_g'', the result is used to seed the summary of the callee procedure g . The condition CP4 is the most complex clause of the proof rule and ensures procedure-modular reasoning. It uses a query from the caller $T_f(V_f, V_f')$, links the calling context with

the parameter passing relation $call_{f,g}(V_f', V_g'')$, uses the summary of the callee $T_g(V_g'', V_g''')$, passes the return value back in the scope of the caller with $ret_{f,g}(V_g''', V_f'''')$ and links local variables not affected by the call with $loc_f(V_f', V_f'''')$. Finally, the condition CP5 requires that states reachable at an arbitrary location in some procedure f do not intersect error states.

Termination For proving termination properties, we use the proof rule from Figure 5 with an additional well-foundedness condition that no infinite recursion is feasible. This condition is imposed on a transition relation that describes recursive descent by composing procedure summaries with call relations following [12]. Let $V_P = \bigcup_{f \in Procs(P)} V_f$ be the set of variables that occur in all procedures. The technique of [12] constructs an assertion $Descent(V_P, V_P')$ such that the program P terminates if and only if $Descent(V_P, V_P')$ is well-founded. By taking a transitive closure of $Descent(V_P, V_P')$ as described above, we can replace well-foundedness condition by a disjunctive well-foundedness condition.

5.3 Multi-threaded programs

We consider a multi-threaded program that consists of N threads as a tuple $(V, init, \rho_1, \ldots, \rho_N)$, where ρ_i is the transition relation of the thread i. The transition relation of the program ρ is the disjoint union of the N transition relations of the threads.

Owicki-Gries rule for proving safety Based on Owicki-Gries method [36], we present the following proof rule for verifying safety of multi-threaded programs. This proof rule lists conditions over N query symbols R_i that characterize reachable states for each thread $i \in 1..N$.

For assertions $R_1, \ldots, R_N$ over V ,

CO1 :	$init(V)$	$\rightarrow R_i(V)$
CO2 :	$R_i(V) \wedge \rho_i(V, V')$	$\rightarrow R_i(V')$
CO3 :	$R_i(V) \wedge (\bigvee_{i \in 1..N \setminus \{j\}} R_j(V) \wedge \rho_j(V, V'))$	$\rightarrow R_i(V')$
CO4 :	$R_1(V) \wedge \ldots R_N(V) \wedge error(V)$	$\rightarrow false$

multi-threaded program P is safe

The conditions CO1, CO2 and CO4 resemble those from the invariance proof rule, while the additional condition CO3 ensures that the query $R_i(V, V')$ corresponding to thread i is free from interference from the transitions of other threads j . The presence of distinct query symbols for the reachable states of each thread allows our HSF algorithm to perform abstraction at the thread boundaries and leads to more scalable verification compared to the monolithic proof rule.

Rely-guarantee rule for proving safety As an alternative to the above proof rule, we can use a proof rule based on rely-guarantee reasoning method [25]. The proof rule uses assertions R_i and E_i that characterize reachable states of each thread $i \in 1..N$ and environment transitions of each thread $i \in 1..N$, respectively, following [19].

Each assertion $R_i(V)$ includes the initial states due to CM1. States reachable after executing a transition of thread i or an environment transition $E_i(V, V')$ are in $R_i(V')$ due to CM2 and CM4 , where $\rho_i^=$ requires that the local variables of thread i do not change during an environment step of thread i . CM3 requires that every step of thread i starting from a reachable state is captured by the environment transitions of each other thread j. CM5 ensures that the intersection of reachable states and error states is empty.

For assertions $R_1, \ldots, R_N$ over V and $E_1, \ldots, E_N$ over V, V' ,

CM1 :	$init(V)$	$\rightarrow R_i(V)$
CM2 :	$R_i(V) \wedge \rho_i(V, V')$	$\rightarrow R_i(V')$
CM3 :	$(\bigvee_{i \in 1..N \setminus \{j\}} R_i(V) \wedge \rho_i(V, V'))$	$\rightarrow E_j(V, V')$
CM4 :	$R_i(V) \wedge E_i(V, V') \wedge \rho_i^=(V, V')$	$\rightarrow R_i(V')$
CM5 :	$R_1(V) \wedge \cdots \wedge R_N(V) \wedge error(V)$	$\rightarrow false$

multi-threaded program P is safe

Rely-guarantee rule for proving termination Rely-guarantee reasoning [25] can be combined with the transition invariance proof rule [37] to prove termination properties of multi-threaded programs. The proof rule uses assertions T_i for transition invariants and E_i for environment transitions [39].

For assertions $T_1, \ldots, T_N$ and $E_1, \ldots, E_N$ over V, V' ,

CG1 :	$init(V) \wedge \rho_i(V, V')$	$\rightarrow T_i(V, V')$
CG2 :	$T_i(V, V') \wedge \rho_i(V', V'')$	$\rightarrow T_i(V', V'')$
CG3 :	$T_i(V, V') \wedge \rho_i(V', V'')$	$\rightarrow T_i(V, V'')$
CG4 :	$(\bigvee_{j \in 1..N \setminus \{i\}} init(V) \wedge \rho_j(V, V'))$	$\rightarrow E_i(V, V')$
CG5 :	$(\bigvee_{j \in 1..N \setminus \{i\}} T_i(V, V') \wedge \rho_j(V', V''))$	$\rightarrow E_i(V', V'')$
CG6 :	$init(V) \wedge E_i(V, V') \wedge \rho_i^=(V, V')$	$\rightarrow T_i(V, V')$
CG7 :	$T_i(V, V') \wedge E_i(V', V'') \wedge \rho_i^=(V, V')$	$\rightarrow T_i(V', V'')$
CG8 :	$T_i(V, V') \wedge E_i(V', V'') \wedge \rho_i^=(V, V')$	$\rightarrow T_i(V, V'')$
CG9 :	$dwf(T_1(V, V') \wedge \cdots \wedge T_N(V, V'))$	

multi-threaded program P terminates

The conditions CG1 and CG2 require that T_i over-approximates the transition relation of the thread i restricted to initial states and to arbitrary reachable states. The condition CG3 extends T_i with a relation from ρ_i . The conditions CG4 and CG5 populate the environment transitions of thread i with steps of all threads other than i . The three conditions CG6, CG7 and CG8 are similar to the first three conditions except local transitions from ρ_i are replaced by environment transitions $E_i \wedge \rho_i^=$. As before, $\rho_i^=$ requires that the local variables of thread i do not change during an environment step of thread i . The final condition CG9 ensures that the conjunction of the transition invariant queries, $T_1(V, V') \wedge \cdots \wedge T_N(V, V')$, is disjunctively well-founded.

6. Experiments

In this section we present an experimental comparison between verifiers developed using our HSF algorithm and state-of-the-art verification tools developed using traditional methods.

Our tool HSF is implemented in Prolog and compiled with the SICStus Prolog 4.2.0 compiler. The implementation relies on a built-in constraint solver for linear arithmetic. For C programs, we use the CIL library [35] and an additional frontend step that produces clauses for various proof rules. For verifying OCaml programs, we use DSolve [43] to generate automatically subtyping constraints and then our code translates these constraints to Horn-like clauses.

Benchmarks We used several sets of benchmarks for our experiments.

Program	BLAST	CPAchecker	HSF
Numerical Recipes			
amebsa	FAIL	T/O	**0.2s**
amotsa	FAIL	2.3s	**0.2s**
bandec	T/O	T/O	T/O
choldc	14.1s	2.7s	**2.6s**
crank	6.0s	1.9s	**0.9s**
cyclic	FAIL	T/O	**2.2s**
four1	FAIL	T/O	T/O
lop	FAIL	FAIL	**0.7s**
pzextr	11.2s	FAIL	**0.1s**
qrdcmp	75.2s	T/O	**12.2s**
qrsolv	FAIL	3.3s	**0.1s**
rsolv	T/O	33.6s	**0.2s**
spline	FAIL	1.6s	**0.3s**
tridag	4.1s	1.5s	**0.3s**
ntdrivers			
cdaudio_simpl1	64.8s	**24.9s**	393s
diskperf_simpl1	41.9s	**21.1s**	213s
floppy_simpl3	30.9s	**10.3s**	67s
floppy_simpl4	50.1s	**16.3s**	139s
kbfiltr_simpl1	3.7s	**2.7s**	3.2s
kbfiltr_simpl2	5.5s	**4.0s**	7.2s
cdaudio_simpl1_BUG	29.3s	**12.3s**	351s
floppy_simpl3_BUG	**1.5s**	7.5s	96s
floppy_simpl4_BUG	**1.5s**	12.9s	135s
kbfiltr_simpl2_BUG	**3.1s**	3.2s	14.7s
ssh-simplified			
s3_clnt_1	103s	8.0s	**7.4s**
s3_clnt_2	147s	59s	**4.2s**
s3_clnt_3	FAIL	7.4s	**7.3s**
s3_clnt_4	80s	9.7s	**6.6s**
s3_srvr_1	FAIL	23.2s	**9.8s**
s3_srvr_2	FAIL	40.0s	**10.1s**
s3_srvr_3	FAIL	**9.9s**	36.1s
s3_srvr_4	FAIL	11.3s	**8.9s**
s3_srvr_6	110s	**41.9s**	49.5s
s3_srvr_7	FAIL	**14.0s**	133s
s3_srvr_8	41.5s	**11.1s**	23.1s
s3_clnt_1_BUG	4.5s	3.0s	**1.3s**
s3_clnt_2_BUG	4.9s	2.6s	**1.4s**
s3_clnt_3_BUG	4.9s	2.9s	**1.3s**
s3_clnt_4_BUG	4.6s	3.0s	**1.4s**
s3_srvr_1_BUG	FAIL	**2.6s**	3.1s
s3_srvr_2_BUG	66s	2.5s	**2.2s**

Program	Threader	HSF
Multi-threaded programs		
Fig2-cex-BUG	0.2s	**0.1s**
Fig2-fixed	0.8s	**0.7s**
Fig4-cex-BUG	4.5s	**0.5s**
Fig4-fixed	1.5s	**0.5s**
Bluetooth2	29.1s	**12.9s**
Bluetooth2-fixed	3.7s	**0.2s**
Bluetooth3-fixed	135s	**18.6s**
Scull	129s	**11.6s**
Dekker	11.1s	**4.0s**
Peterson	4.7s	**3.7s**
Readers-writer-lock	0.2s	**0.1s**
Time varying mutex	11.8s	**9.8s**
Szymanski	32s	**8.7s**
NaiveBakery	**2.5s**	2.6s
Bakery	105s	**32.4s**
Lamport	121s	**30.5s**
QRCU	34.5s	**15.4s**

Program	HMC	HSF
OCaml programs (correct / buggy)		
na_dotprod-m	**0.04s / 0.04s**	0.11s / 0.06s
na_arraymax-m	0.32s / **0.05s**	**0.05s** / 0.07s
na_bcopy-m	0.09s / 5.94s	**0.06s / 0.07s**
na_bsearch-m	0.91s / 0.10s	**0.03s / 0.02s**
na_insertsort-m	**0.03s / 0.03s**	1.78s / 0.04s
mult-cps-m	**0.03s / 0.03s**	**0.03s / 0.03s**
mult-all-m	0.03s / 0.03s	**0.01s / 0.01s**
sum-all-m	0.03s / 0.03s	**0.01s / 0.01s**
sum-acm-m	0.04s / 0.03s	**0.01s / 0.01s**

Program	HSF
Terminating loops	
broydn (33 cutpoints)	591s
elmhes (9 cutpoints)	23.0s
jacobi (15 cutpoints)	16.0s
ludcmp (11 cutpoints)	4.2s
qrdcmp (9 cutpoints)	189s
rlft3 (7 cutpoints)	16.1s
spctrm (14 cutpoints)	86s

Table 1. Timings for the benchmarks. The left-side of the page shows statistics for sequential programs, with multi-threaded programs, functional programs and terminating loops on the right-side of the page. "T/O" stands for time out after 10 minutes, while "FAIL" indicates that the tool failed to return a verification result.

For verification of sequential programs, we used a set of programs from the Numerical Recipes book [40], with array bound checking being the safety property to verify. This set of benchmarks includes simulated annealing (amebsa), evaluate a trial point using simulated annealing (amotsa), fast fourier transform (four1), or polynomial extrapolation (pzextr). We also used two sets of benchmarks (ntdrivers and ssh-simplified) from the test suite of CPAchecker [7].

We collected multi-threaded and functional programs from the test suites of specialized verification tools, i.e., Threader [18] and HMC [24]. For termination checking, we used a set of programs from the Numerical Recipes book, including a secant method program (broydn), and a program to reduce a matrix to the Hessenberg form (elmhes).

Our benchmark suite includes both safe and non-safe programs. Non-safe programs have the word "BUG" attached to their name. For termination checking, all the benchmarks are terminating and

we report the number of cutpoints as a measure for the effort to verify program termination.

Evaluation Our experiments were run on an Intel Core 2 Duo machine, clocked at 3.0 GHz, with 4 GB of RAM, and running Linux 2.6.38. See Table 1 for the results.

For the first three categories of benchmarks (Numerical Recipes, ntdrivers and ssh-simplified), we used HSF with the summarization proof rule described in Section 2. For the verification of these sequential C programs, we compare HSF with BLAST [20, 21] and CPAchecker [7]. For ntdrivers and ssh-simplified, we used BLAST 2.5 with the MathSat solver [9] and the following standard options: `-craig 2 -dfs -predH 7 -nosimplemem -alias ""`, as suggested by the tool's authors. The use of the MathSat solver led to Blast failing all the Numerical Recipes benchmarks, so instead we report statistics using the latest publicly available version of Blast that uses the Simplify solver. For CPAchecker, we used the revision r3842 from the tool repository and used the predicate abstraction with large block encoding configuration as suggested by the tool's authors.

For verification of multi-threaded programs, we used HSF with a proof rule based on rely-guarantee reasoning [25]. We compare HSF with Threader using a reasoning style that is equivalent to the proof rule mentioned above.

For the verification of functional OCaml programs, we compare HSF with HMC [24]. The techniques used in HMC extend a liquid type system (i.e., [43] that requires user-provided logical qualifiers) to enable automatic verification of OCaml programs.

We used HSF with a termination proof rule for the programs from the last table, Terminating loops. We do not have access to any public tool that can handle termination properties for these C benchmarks.

In general, HSF is comparable and sometimes significantly faster than state-of-the-art tools specialized to a particular verification proof rule. For all our experiments, the verification tools (including HSF) were run starting with an empty set of predicates, i.e., all predicates needed for verification were discovered automatically. Two limitations of our implementation lead to HSF being slower for the ntdrivers benchmarks: no direct support for equality predicates and the program representation with the transition relation in disjunctive normal form. For example, instead of a single equality predicate (say $x = y$), HSF tracks two predicates, i.e., $x \geq y$ and $x \leq y$, both during abstraction and refinement. We plan to implement heuristics to handle the high level of branching present in this set of benchmarks. On the other hand, we leverage the uniform representation of the Horn clauses for simplification and inlining steps before the start of the verification process. These transformations lead to substantial savings that are particularly effective in the Numerical Recipes, multi-threaded, and OCaml benchmarks. In total, HSF took approximately 48 minutes to analyze 35 kloc (we exclude the lines of code for the two programs on which HSF timed out).

We applied our tool on benchmarks from various classes of verification problems, which are usually approached using specialized tools.

Verification competition HSF(C) is a verifier for C programs developed using the HSF algorithm and based on the summarization proof rule. HSF(C) participated in the TACAS2012 software verification competition [6] and reached the 3rd place in the largest category ControlFlowInteger and it competed with recent implementations of Blast, CPAchecker and 6 more verification tools. (HSF(C) did not participate in the other categories that required bit-precise reasoning or pointer analysis features not supported by our CIL frontend.)

For the ControlFlowInteger category, HSF(C) analyzed 96 benchmarks from five groups: locks, ntdrivers, ntdrivers-simplified, ssh and ssh-simplified. Each of these benchmarks consists of a C program and a safety property, which may or may not hold. HSF timed out on 2 of these programs, verifying and finding counterexamples correctly for all the others, in total 207.2 kloc analyzed in 80 minutes. More details about HSF(C) can be found in the related competition report [16].

7. Related work

Section 1 points to existing approaches to generate various analyses based on dataflow domains that led to powerful program analysis frameworks [3, 27–29, 31, 34, 47, 51].

Verifiers have also been a target for automated tool construction. XSB [41] is a programmable fixed-point engine used for implementing model checkers for a concurrent language based on CCS with properties specified in a fragment of mu-calculus. Model checkers have been generated from algebraic specifications of a source language and various fragments of temporal logic [45]. More recently, verifier generators have been developed for Boolean programs (GETAFIX [49]) and programs for which Datalog style bottom up inference terminates (μZ [22]). For programs with unbounded data domains, MatchC [44] provides a verifier based on matching logic specifications that directly build upon the operational semantics of the source language. The verification is facilitated by (pattern) loop invariants provided by the programmer. In comparison, our approach adds an abstraction refinement loop, which is crucial for handling unbounded datatypes, and allows automation of proof rules for termination and liveness properties.

HSF synthesizes verifiers that are competitive with state-of-the-art tools from a recent verification competition, while benefiting from a series of verification algorithms. We build upon predicate abstraction [15], counterexample guided abstraction refinement [10], interpolation [32], ranking function generation [8, 38], and constraint solvers for recursion-free Horn clauses [17, 19, 39].

Our work provides an intermediate representation for verification tasks in the form of Horn-like clauses with the support for a dwf-predicate. This representation was inspired by the usage of Horn clauses for the inference of environment assumptions of multi-threaded programs [19]. Boogie provides a different intermediate representation for procedural and object-oriented programs. Boogie represents programs and therefore it requires an intermediate step to generate verification conditions, while Horn clauses encode verification tasks directly as constraints. We believe that generation of Horn clauses from Boogie programs will yield yet another verification tool for Boogie and all of its input languages.

8. Conclusion

We presented a next logical step towards the automatic generation of software verification tools. Our verifier generator takes as input a proof rule written as Horn-like clauses and produces a verifier that automates the proof rule. The experimental evaluation shows that automatically generated verifiers are competitive with existing state-of-the-art verification tools that are manually developed and tuned.

Acknowledgments

We thank Jasmin Blanchette for comments and suggestions.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2006.

[2] A. Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2), 1999.

[3] A. Aiken, M. Fähndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Types in Compilation*, 1998.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[5] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *TACAS*, 2001.

[6] D. Beyer. Competition on software verification - (SV-COMP). In *TACAS*, 2012.

[7] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, 2011.

[8] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, 2005.

[9] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4SMT solver. In *CAV*, 2008.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.

[11] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[12] B. Cook, A. Podelski, and A. Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3), 2009.

[13] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, 2003.

[14] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

[15] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.

[16] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, 2012.

[17] A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *APLAS*, 2011.

[18] A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV*, 2011.

[19] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.

[22] K. Hoder, N. Bjørner, and L. de Moura. μZ- an efficient engine for fixed points with constraints. In *CAV*, 2011.

[23] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software verification platform. In *CAV*, 2005.

[24] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *CAV*, 2011.

[25] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4), 1983.

[26] K. W. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP*, 2007.

[27] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, 2005.

[28] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS*, 2005.

[29] S. Lerner, T. D. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

[30] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. 1995.

[31] F. Martin. PAG – An efficient program analyzer generator. *STTT*, 2 (1), 1998.

[32] K. L. McMillan. An interpolating theorem prover. *TCS*, 2005.

[33] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[34] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[35] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, 2002.

[36] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6, 1976.

[37] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.

[38] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

[39] C. Popeea and A. Rybalchenko. Compositional termination proofs for multi-threaded programs. In *TACAS*, 2012.

[40] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. 1992.

[41] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV*, 1997.

[42] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[43] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[44] G. Rosu and A. Stefanescu. Matching logic: a new program verification approach. In *ICSE*, 2011.

[45] T. Rus, E. V. Wyk, and T. Halverson. Generating model checkers from algebraic specifications. *Formal Methods in System Design*, 20(3), 2002.

[46] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, 2007.

[47] M. Sagiv. *High Level Formalisms for Program Flow Analysis and their use in Compiling*. PhD thesis, Technion, 1991.

[48] T. Terauchi. Dependent types from counterexamples. In *POPL*, 2010.

[49] S. L. Torre, P. Madhusudan, and G. Parlato. Analyzing recursive programs using a fixed-point calculus. In *PLDI*, 2009.

[50] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP*, 2009.

[51] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.