Conflict-Driven Conditional Termination

Vijay d'Silva, Caterina Urban

Conflict-Driven Conditional Termination

Vijay D'Silva[1] and Caterina Urban[2]

[1] Google Inc., San Francisco
[2] École Normale Supérieure, Paris

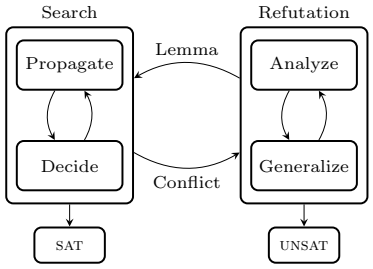
Abstract. Conflict-driven learning, which is essential to the performance of SAT and SMT solvers, consists of a procedure that searches for a model of a formula, and refutation procedure for proving that no model exists. This paper shows that conflict-driven learning can improve the precision of a termination analysis based on abstract interpretation. We encode non-termination as satisfiability in a monadic second-order logic and use abstract interpreters to reason about the satisfiability of this formula. Our search procedure combines decisions with reachability analysis to find potentially non-terminating executions and our refutation procedure uses a conditional termination analysis. Our implementation extends the set of conditional termination arguments discovered by an existing termination analyzer.

1 Conflict-Driven Learning for Termination

Conflict-driven learning procedures are integral to the performance of SAT and SMT solvers. Such procedures combine search and refutation to determine if a formula is satisfiable. Conflicts discovered by search drive refutation, and search learns from refutation to avoid regions of the search space without solutions.

Our work is driven by the observation that discovering a small number of disjunctive termination arguments is crucial to the performance of certain termination analyzers [**?**]. Fig. **??** summarizes our lifting of conflict-driven learning to termination analysis. We use reachability analysis to find a set of states that constitute potentially non-terminating execution. We apply a conditional termination analysis to this set to eliminate states from which all executions terminate. Unlike termination analysis, which solves a decision problem and returns a YES or NO answer, conditional termination analysis is concerned with discovering sufficient conditions for termination. Sufficient conditions for termination play the role of learned clauses in our analysis. They prevent subsequent runs of reachability analysis from revisiting states from which termination is guaranteed.

Our conflict driven conditional termination procedure (CDCT) can be viewed as a sound but incomplete solver for a family of monadic, second-order formulae. Büchi's theorem shows that the language of a Büchi automaton is non-empty exactly if a formula in the monadic second-order theory of one successor (S1S) is satisfiable [**?**]. This theorem can be viewed encoding non-termination of a finite-state program as satisfiability in S1S. We introduce S1S(T), an extension of S1S to sequences of first-order structures, and encode non-termination in a



SAT/SMT	Termination
Satisfiability	Non-Termination
Model	Infinite execution
Countermodel	Finite or infeasible trace
Constraint Propagation	Reachability Analysis
Conflict Analysis	Termination Analysis

Fig. 1: Conflict Driven Learning as applied to Termination

control-flow graph (CFG) as satisfiability in S1S(T). A model of a formula is an infinite execution that respects the transition constraints in the CFG.

Formulating non-termination as satisfiability provides a clear route for lifting CDCL to non-termination. We combine decisions with reachability in an abstract domain to construct and refine assignments to second-order variables in the same way that SAT solvers construct and refine partial assignments. A notable difference to standard abstract interpretation is that our assignments are neither over- nor under-approximations of the set of reachable states. Our conflict analysis uses backwards abstract interpretation to enlarge the set of states from which termination is guaranteed. We present a generalized unit rule for combining ranking functions with reachability analysis. These components are combined in our new analysis, which we have implemented and evaluated against state-of-the-art termination provers.

2 Non-Termination as Second-Order Satisfiability

The two contributions of this section are the logic S1S(T), which extends the monadic second-order logic of one successor (S1S) with a theory and an encoding of program non-termination as satisfiability in this theory.

2.1 Monadic Second-Order Theories of One Successor

We use $\hat{=}$ for definition. Let $\mathcal{P}(S)$ be the powerset of S. For $f : A \to B$, the function $f[a \mapsto b]$ maps a to b and c distinct from a to $f(c)$. The symbols x, y, z range over first-order variables in *Vars*, f, g, h over functions in *Fun*, and P, Q, R over predicates in *Pred*. We use a set *Pos* of first-order *position variables* whose elements are i, j, k, a set *SVar* of monadic second-order variables denoted X, Y, Z, a unary successor function *suc* and a binary successor predicate *Suc*.

Our logic consists of three families of formulae called state, transition and trace formulae, which are interpreted over first-order structures, pairs of first-order structures and infinte sequences of first-order structures respectively. The

formulae are named after how they are interpreted over programs.

$$
\begin{array}{rr}
t ::= x \mid f(t_0, \ldots, t_n) & \text{Term} \\
\varphi ::= P(t_0, \ldots, t_n) \mid \varphi \wedge \varphi \mid \neg\varphi & \text{State Formula} \\
\psi ::= suc(x) = t \mid \psi \wedge \psi \mid \neg\psi & \text{Transition Formula} \\
\Phi ::= X(i) \mid Suc(i,j) \mid \varphi(i) \mid \psi(i) & \\
\quad \mid \Phi \wedge \Phi \mid \neg\Phi \mid \exists i : Pos.\Phi & \text{Trace formula}
\end{array}
$$

A first-order interpretation (Val, I) defines functions $I(f)$ and relations $I(P)$ over values in Val. The value $[\![t]\!]_s$ of a term t in a *state* $s : Vars \to Val$, is $s(x)$ if t is x, and $I(f)([\![t_0]\!]_s, \ldots, [\![t_n]\!]_s)$ if t is $f(t_0, \ldots, t_n)$. The interpretation of a state formula is the standard first-order semantics. A transition formula is interpreted at a *transition*, that is, a pair of states (r, s). A formula φ in which the symbol suc does not occur is interpreted at the state r, while $suc(x) = t$ compares the value of the term t in r with the value of x in the successor state s.

$$
\begin{array}{ll}
(r,s) \models P(t_0, \ldots, t_n) \text{ if } ([\![t_0]\!]_r, \ldots, [\![t_n]\!]_r) \in I(P) & \\
(r,s) \models \varphi \wedge \psi \text{ if } (r,s) \models \varphi \text{ and } (r,s) \models \psi & \\
(r,s) \models \neg\varphi \text{ if } (r,s) \not\models \varphi & (r,s) \models suc(x) = t \text{ if } [\![x]\!]_s = [\![t]\!]_r
\end{array}
$$

A *trace* $\tau : \mathbb{N} \to (Vars \to Val)$ is an infinite sequence of states and $\tau(m)$ is the *state at position* m. A *position assignment* $\sigma : (Pos \to \mathbb{N}) \cup (SVar \to \mathcal{P}(\mathbb{N}))$ maps position variables to $\mathbb{N}$ and second-order variables to subsets of $\mathbb{N}$ such that $\{\sigma(X) \mid X \in SVar\}$ partitions $\mathbb{N}$. We explain this partition condition shortly. A trace formula is interpreted with respect to an S1S(T) *structure* (τ, σ).

Note that there are first-order variables of two sorts in a trace formula. A trace formula Φ asserting that the transition formula $\psi(x,y) \mathrel{\hat{=}} suc(x) = y + 1$ is true at the trace position denoted by i has the form $\psi(x,y)(i)$. The predicate $Suc(i,j)$ asserts that the position j occurs immediately after i.

$$
\begin{array}{ll}
(\tau,\sigma) \models Suc(i,j) \text{ if } \sigma(i) + 1 = \sigma(j) & (\tau,\sigma) \models \varphi(i) \text{ if } \tau(\sigma(i)) \models \varphi \\
(\tau,\sigma) \models \psi(i) \text{ if } (\tau(\sigma(i)), \tau(\sigma(i)+1)) \models \psi & (\tau,\sigma) \models X(i) \text{ if } \sigma(i) \in \sigma(X) \\
(\tau,\sigma) \models \Phi \wedge \Psi \text{ if } (\tau,\sigma) \models \Phi \text{ and } (\tau,\sigma) \models \Psi & (\tau,\sigma) \models \neg\Phi \text{ if } (\tau,\sigma) \not\models \Phi \\
(\tau,\sigma) \models \exists i : Pos.\Phi \text{ if } (\tau, \sigma[i \mapsto n]) \models \Phi & \text{for some } n \text{ in } \mathbb{N}
\end{array}
$$

An S1S(T) structure (τ, σ) is a *model* of Φ if $(\tau, \sigma) \models \Phi$, and is a *countermodel* otherwise. A trace formula is *satisfiable* if it has a model. An S1S(T) structure is defined using an infinite trace, so finite traces cannot be models of a formula.

2.2 Encoding Non-Termination in S1S(T)

We now recall control flow graphs (CFGs) and encode non-termination as satisfiability. A *command* in Cmd is an assignment $x := t$ of a term t to a first-order variable x, or is a condition $[\varphi]$, where φ is a state formula. A CFG $G = (Loc, E, \mathtt{in}, \mathtt{ex}, stmt)$ consists of a finite set of locations Loc including an

$$
\begin{array}{c}
(\forall i.\mathit{First}(i) \Rightarrow X_{\mathtt{in}}(i)) \wedge (\forall i.X_{\mathtt{ex}}(i) \Rightarrow \mathit{Last}(i)) \\
\wedge\ \forall i.\forall j.X_{\mathtt{in}}(j) \wedge \mathit{Suc}(i,j) \Rightarrow (\mathit{suc}(x) = x - 1)(i) \wedge X_{\mathtt{a}}(i) \\
\wedge\ \forall i.\forall j.X_{\mathtt{a}}(j) \wedge \mathit{Suc}(i,j) \ \Rightarrow (x \neq 0 \Rightarrow \mathit{suc}(x) = x)(i) \wedge X_{\mathtt{in}}(i) \\
\wedge\ \forall i.\forall j.X_{\mathtt{ex}}(j) \wedge \mathit{Suc}(i,j) \Rightarrow (x = 0 \Rightarrow \mathit{suc}(x) = x)(i) \wedge X_{\mathtt{in}}(i)
\end{array}
$$

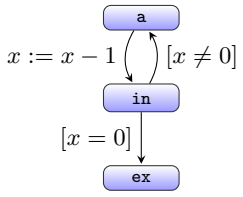


Fig. 2: A formula encoding non-termination of the program shown in the monadic second-order theory of one successor over integer arithmetic.

initial location $\mathtt{in}$, an exit location $\mathtt{ex}$, edges $E \subseteq \mathit{Loc} \times \mathit{Loc}$, and a labelling $\mathit{stmt} : E \to \mathit{Cmd}$ of edges with commands. To assist the presentation, we assume that the exit location $\mathtt{ex}$ has no successors.

The formula Trans_c below defines the semantics of commands using the condition $\mathit{Same}_V \mathrel{\hat{=}} \bigwedge_{x \in V} \mathit{suc}(x) = x$, that variables in V are not modified. The set of models of Trans_c is the transition relation Rel_c. We write Trans_e and Rel_e for the transition formula and relation of the command $\mathit{stmt}(e)$. The formula Inf_G extends the translation of Büchi automata to S1S to encode CFGs in S1S(T). We write $\mathit{First}(i) \mathrel{\hat{=}} \forall j.\neg \mathit{Suc}(j,i)$ for the first position on a trace and $\mathit{Last}(i) \mathrel{\hat{=}} \forall j.\neg \mathit{Suc}(i,j)$ for a position that cannot be on an infinite trace.

$$
\mathit{Trans}_c \mathrel{\hat{=}} \begin{cases} b \implies \mathit{Same}_{\mathit{Vars}} & \text{if } c = [b] \\ \mathit{suc}(x) = t \wedge \mathit{Same}_{\mathit{Vars} \setminus \{x\}} & \text{if } c = x := t \end{cases}
$$

$$
\begin{array}{rl}
\mathit{Inf}_G \mathrel{\hat{=}} & (\forall i.\mathit{First}(i) \implies X_{\mathtt{in}}(i)) \wedge (\forall j.X_{\mathtt{ex}}(j) \implies \mathit{Last}(j)) \\
& \wedge \bigwedge_{\mathtt{v} \in \mathit{Loc}} \forall i.\forall j.X_{\mathtt{v}}(j) \wedge \mathit{Suc}(i,j) \implies \bigvee_{(\mathtt{u},\mathtt{v}) \in E} \mathit{Trans}_{(\mathtt{u},\mathtt{v})}(i) \wedge X_{\mathtt{u}}(i)
\end{array}
$$

The formula Inf_G encodes program behaviour as follows. Consider an S1S(T) structure (τ, σ). The interpretation $\sigma(X_\ell)$ of a second-order variable X_ℓ represents positions on the trace when execution is at location ℓ. Such an interpretation partitions $\mathbb{N}$ because each position on a trace corresponds to a unique location. The *entry constraint* on $\mathit{First}(i)$ ensures execution begins at $\mathtt{in}$. The *exit constraint* implying $\mathit{Last}(j)$ enforces that an infinite execution does not visit $\mathtt{ex}$. The conditions involving $\mathit{Suc}(i,j)$ are called *transition constraints* and express that consecutive states on a trace must respect the transition relation of G. Theorem **??** expresses non-termination as satisfiability.

Theorem 1. *A CFG G has a non-terminating execution iff Inf_G is satisfiable.*

We believe this is a simple yet novel encoding of non-termination that allows the duality between search and refutation to be exploited for termination analysis. In contrast, the second-order encoding of termination in [**?**] uses a predicate for disjunctive well-foundedness and is solved in a different manner.

Example 1. A CFG G and the formula Inf_G for a program with a variable x of type $\mathbb{Z}$ are shown in Fig. **??**. We write a trace as a sequence of values of x. Let τ

be the trace $-1, -1, -2, -2, \ldots$ and σ the assignment mapping $X_{\mathtt{ex}}$ to the empty set, and $X_{\mathtt{in}}$ and $X_{\mathtt{a}}$ to even and odd positions, respectively. The structure (τ, σ) is a model of Inf_G. Every structure (τ, δ), with τ as before, in which $\delta(X_{\mathtt{ex}})$ is not empty is a countermodel of Inf_G because ex is not reachable if x is initially -1, so some transition in τ must violate a transition constraint in Inf_G. Every structure (τ', δ') with x non-negative in $\tau'(0)$ is also a countermodel of Inf_G because executions with x initially non-negative terminate. Since τ' is infinite by definition, some transition in τ' must be infeasible. Terminating executions cannot be models of Inf_G because traces in S1S(T) structures are infinite. $\lhd$

The formula Inf_G is a conjunction of formulae in which second-order variables and first-order program variables are free but first-order position variables are bound. We exploit this structure in our analysis.

3 Conflict-Driven Conditional Termination

The conflict-driven conditional termination procedure (CDCT) in Alg. **??** generalizes CDCL from SAT to termination analysis. The input is the formula Inf_G. The output $(\mathsf{result}, \Delta, \Theta)$ is a result concerning a set of structures Δ and a set Θ of piecewise-defined ranking functions (PDRFs).

The value of result is one of divergent, terminates, or unknown. CDCT returns divergent if the traces represented by Δ do not reach the exit location, which could be due to non-termination or undefined behaviour; It returns terminates if Δ is empty and Θ guarantees termination for all states. It returns unknown if CDCT cannot prove termination and cannot progress. This happens if the abstract domain cannot accurately represent non-terminating executions, if the ranking functions used cannot express a termination argument, or a bound on the number of decisions has been exceeded.

CDCT maintains four global data structures. The *trail* tr is a sequence of assignments to second-order variables. The *explanation array* exp contains in each element $exp[i]$, the decision or constraint used by propagation to add $tr[i]$ to the trail. The set of PDRFs Θ, generated by conditional termination analysis, are our analogue of learned clauses. The *blocking constraints* Ψ contain constraints representing two types of states, which need not be revisited. One is states from which all executions terminate. The other is states for which CDCT could neither prove termination nor demonstrate non-termination.

Each execution of the CDCT loop begins with a call to Search(), which attempts to find a non-terminating execution. If Search() returns divergent, CDCT returns. If Search() returns unknown, the trail represents a *potential conflict* because it has discovered a set of states from which some execution terminates. The conflict is potential because the trail may also contain models of Inf_G. This is a difference to SAT and SMT solvers where a conflict contradicts a formula.

The conflict analysis procedure Analyze() extracts from a potential conflict a definite conflict θ, expressed as a ranking function. The domain of θ represents states from which all executions terminate. The learning step Learn() generates a blocking constraint to drive subsequent search away from these states. Learning

Algorithm 1: CDCT(Inf_G)

Trail: $tr \leftarrow \epsilon$
Explanations: $exp \leftarrow \epsilon$
Blocking constraints: $\Psi \leftarrow \emptyset$
PDRFs: $\Theta \leftarrow \emptyset$
while *true* **do**
  result ← Search()
  if result = divergent *or*
  (result = unknown *and* exceeded()) **then**
    return (result, $[tr], \Theta$)
  $\theta \leftarrow$ Analyze()
  $\Theta \leftarrow \Theta \cup \{\theta\}$
  $\Psi \leftarrow \Psi \cup$ Learn($[tr], \theta$)
  if Backtrack() = *false* **then**
    return (terminates, $[\epsilon], \Theta$)

```
ℤ step(ℤ x) {
  if (x>20)
    return 3;
  else if (x>10)
    return 2;
  else
    return 1;
}
void main() {
    y, i : ℤ
[a] if (y>0)
      i = −step(y);
    else
      i = step(−y);
[b] while(y<−3||y>3)
      y = y+i;
[ex]}
```

also generates a blocking constraint if CDCT cannot make progress analyzing $[tr]$. This happens if no more decisions can be made and no ranking function can be extracted. CDCT then backtracks if possible.

An Example Run. A program is shown in C-like syntax alongside Alg. **??**. The location `a` is reached after the variables are initialized, `b` is the loop head, and `ex` is the exit location. The program terminates but the abstract interpretation-based tool FuncTion [**?**] cannot prove termination. CDCT enables FuncTion to prove termination while also avoiding case explosion. Even though other tools may be able to prove termination, we believe CDCT is interesting because similar ideas could be used to expand the programs handled by those tools.

In this example, we use an interval abstract domain and affine ranking functions. Search() uses reachability analysis to derive the intervals $y{:}[-3,3], i{:}[-3,3]$ at `ex` but termination analysis fails. Decisions restrict the range of a variable at a location: for example, Search() heuristically uses conditions from the code to make the decisions $y{:}[1,\infty]$ and $y{:}[-\infty,10]$ at location `a`. Reachability derives the range $y{:}[1,3], i{:}[-1,-1]$ at `ex`, which is a conflict, because no trace with these states at `ex` satisfies Inf_G. Analyze() represents this conflict as $X_{\texttt{ex}} \mapsto \{y{:}[1,3], i{:}[-1,-1] \to 0\}$, which assigns a PDRF to the second-order variable $X_{\texttt{ex}}$ and expresses that the program terminates in 0 steps for the states shown. The PDRF is propagated backwards through the program by an abstract interpreter [**?**] to derive the second-order assignments below. We omit the interval on i, which is unchanged.

$$X_{\texttt{ex}} \mapsto y{:}[1,3] \to 0, X_{\texttt{b}} \mapsto y{:}[1,3] \to 1, X_{\texttt{b}} \mapsto y{:}[4,4] \to 3, X_{\texttt{b}} \mapsto y{:}[5,5] \to 5$$

If these assignments are propagated to location `b`, we could only prove that the program terminates for $y{:}[1,5]$ at `a`. Instead, we apply widening to the PDRFs to derive $X_b \mapsto \{y{:}[1,3] \to 1, y{:}[4,10] \to 2x+5\}$, which bounds the number of

steps to termination at the loop head for y in the ranges shown. We heuristically expand the piece $y{:}[4, 10]$ of the PDRF to $y{:}[1, \infty]$ and check if the $2x + 5$ is still a ranking function. Since it is, we have proved termination for executions with $y{:}[1, \infty], i{:}[-1, -1]$ at `b`, despite having explicitly only analyzed the range $y{:}[0, 5]$.

The learning step complements the decision $y{:}[1, \infty]$ and uses $X_{\mathtt{a}} \mapsto y{:}[-\infty, 0]$ to restrict future search. Learnt constraints typically have more structure. A similar run of CDCT can show termination when y is initially non-positive.

Consider the program with the loop condition changed to $(y > -3)$. Now, the program does not always terminate. Decisions and learning can infer a ranking function for positive y as before. Decisions can also discover that for $X_{\mathtt{a}} \mapsto y{:}[-1, -1]$, `ex` is unreachable, indicating non-termination (as all locations lead to `ex`). In this way, CDCT proves conditional termination using disjunctions of ranking functions and also identifies non-terminating executions.

4 Search for a Conflict

We now show how a trail, a data structure used by SAT solvers, can be used to make explicit the incremental progress made by an abstract interpreter.

Abstract Domains. A bounded lattice $(L, \sqsubseteq, \sqcap, \sqcup)$ is a partially ordered set with a meet $\sqcap$, a join $\sqcup$, a greatest element $\top$ (top), and a least element $\bot$ (bottom). A concrete domain for forward analysis $(\mathcal{P}(\mathit{State}), \subseteq, F)$ is a lattice of states with a set $F = \{\mathit{post}_c \mid c \in \mathit{Cmd}\}$ of monotone functions called *transformers*, where $\mathit{post}_c(S)$ is the image of S under the transition relation for c. An *abstract domain* is a bounded lattice $(A, \sqsubseteq, G, \bigtriangledown)$ with a set of abstract transformers $G = \left\{\mathit{post}^A_c \mid c \in \mathit{Cmd}\right\}$ and a *widening operator* $\bigtriangledown : A \times A \to A$. There is a monotone *concretization* function $\gamma : A \to \mathcal{P}(\mathit{State})$ satisfying that $\gamma(\top) = \mathit{State}$ and $\gamma(\bot) = \emptyset$. The transformers satisfy the soundness condition $\mathit{post}_c(\gamma(a)) \subseteq \gamma(\mathit{post}^A_c(a))$ that abstract transformers overapproximate concrete transformers.

Literals are essential for propagation and conflict analysis in SAT. The analogue of literals in abstract domains are complementable meet-irreducibles [**?**]. A lattice element c is a *meet-irreducible* if $a \sqcap b = c$ implies that $a = c$ or $b = c$. Let $\mathcal{M}_A$ be the meet-irreducibles of A. An abstract element a has a *concrete complement* if there exists an $\overline{a}$ in A such that $\gamma(a) = \neg\gamma(\overline{a})$. A *meet decomposition* of an element a is a finite set $\mathit{mdc}(a) \subseteq \mathcal{M}_A$ satisfying that $\bigsqcap \mathit{mdc}(a) = a$ and that there is no strict subset $S \subset \mathit{mdc}(a)$ with $\bigsqcap S = a$. A has *complementable meet irreducibles* if every $m \in \mathcal{M}_A$ has a concrete complement $\overline{m} \in \mathcal{M}_A$.

Example 2. The interval lattice has elements $[a, b]$, where $a \leq b \in \mathbb{Z} \cup \{-\infty, \infty\}$. The intervals $[-\infty, k], [k, \infty]$ are meet-irreducibles, unlike $[0, 2]$. The set $S = \{[-\infty, 2], [0, \infty], [-5, \infty]\}$ satisfies $\bigsqcap S = [0, 2]$ but is not a meet decomposition because $\{[-\infty, 2], [0, \infty]\} \subset S$. The concrete complements of $[-\infty, k]$ and $[k, \infty]$ are $[k + 1, \infty]$ and $[-\infty, k - 1]$, while $[0, 2]$ has no concrete complement. $\lhd$

Abstract Assignments. SAT solvers use partial assignments to incrementally construct a model. We introduce abstract assignments, which use abstract domains

Algorithm 2: Search()

```
while true do
    Propagate()
    if tr(X_ex) = ⊥ then
        return divergent
    d ← dec(Inf_G, Ψ, tr)
    if [tr] ⊑ [tr·d] then
        return unknown
```

	Trail tr	exp	Modification
1	ϵ		Initial state
2	$X_{\mathtt{ex}}$:$[-\infty, 0]$, $X_{\mathtt{ex}}$:$[0, \infty]$	$\{\mathtt{in}, \mathtt{a}, \mathtt{ex}\}$	Propagation
3a	$\hookrightarrow X_{\mathtt{in}}$:$[9, \infty]$ $X_{\mathtt{in}}$:$[0, \infty]$	dec	Decision
4a	X_a:$[1, \infty]$	$\{\mathtt{a}, \mathtt{in}\}$	Propagation
5a	$X_{\mathtt{in}}$:$[-\infty, 0]$	dec	Decision
6a	$X_{\mathtt{a}}$:$\bot$	$\{\mathtt{a}, \mathtt{in}\}$	Propagation
3b	$X_{\mathtt{in}}$:$[-\infty, -7]$	dec	Decision
4b	X_a:$[-\infty, -7]$, $X_{\mathtt{ex}}$:$\bot$	$\{\mathtt{a}, \mathtt{in}\}$	Propagation

to represent S1S(T) structures. Let *Struct* be the set of S1S(T) structures. The *lattice* of *abstract assignments* $(Asg_A, \sqsubseteq)$ contains the set $Asg_A \mathrel{\hat{=}} SVar \to A$ with the *pointwise order*: $asg \sqsubseteq asg'$ if $asg(X) \sqsubseteq asg'(X)$ for all X in $SVar$. The meet and join are also defined pointwise. An abstract assignment asg represents a set of S1S(T) structures as defined by the concretization $conc : Asg_A \to \mathcal{P}(Struct)$.

$$conc(asg) \mathrel{\hat{=}} \{(\tau, \sigma) \mid \text{for all } X \in SVar.\, \{\tau(i) \mid i \in \sigma(X)\} \subseteq \gamma(asg(X))\}$$

An abstract assignment asg is a *definite conflict* for Φ if no model of Φ is in $conc(asg)$ and is a *potential conflict* if $conc(asg)$ contains a countermodel of Φ.

Trail. We introduce a trail, which contains meet-irreducibles as in [**?**,**?**] and in which a second-order variable can appear multiple times. A *trail* over A is the empty sequence ϵ or the concatenation $tr{\cdot}(X{:}m)$, where X is a second-order variable and m is a complementable meet-irreducible. A trail tr defines the assignment $[tr]$ where $[\epsilon] \mathrel{\hat{=}} \lambda Y.\top$ and $[tr{\cdot}(X{:}m)]$ maps X to $[tr](X) \sqcap m$ and all other Y to $[tr](Y)$. A trail tr is in potential/definite conflict with Φ if $[tr]$ is. We write $tr(X)$ for $[tr](X)$. An *explanation* exp for a trail of length n is a function from $[0, n-1]$ to constraints in Inf_G or learnt clauses.

Search(). Alg. **??** extends a trail tr by propagating constraints from the CFG, making decisions, or applying a generalized unit rule. It returns divergent if $tr(X_{\mathtt{ex}})$ is $\bot$, meaning that ex is unreachable. It returns unknown if $tr(X_{\mathtt{ex}})$ is not $\bot$ and no decisions can be made. This trail is a potential conflict because every structure in $conc([tr])$ with a non-empty assignment to $X_{\mathtt{ex}}$ violates the constraint $X_{\mathtt{ex}}(i) \implies Last(i)$, hence is a countermodel of Inf_G.

Example 3. The table alongside Alg. **??** illustrates the construction of tr and exp during interval analysis of the program in Fig. **??**. The exp column shows the locations of the propagated constraints. The rows $1, 2, 3a, 4a, 5a, 6a$ represent a run of Search(). The trail is initially empty and the result of standard interval analysis is the trail $X_{\mathtt{ex}}$:$[-\infty, 0]$, $X_{\mathtt{ex}}$:$[0, \infty]$ in step 2, representing the assignment $\{X_{\mathtt{in}} \mapsto \top, X_{\mathtt{a}} \mapsto \top, X_{\mathtt{ex}} \mapsto [0, 0]\}$. An arbitrary decision $X_{\mathtt{in}}$:$[9, \infty]$ in step $3a$ is not sound (see Ex. **??**) and the smallest sound decision containing it is $[0, \infty]$. Propagation yields $X_{\mathtt{a}}$:$[1, \infty]$ in step $4a$. The decision $X_{\mathtt{in}}$:$[-\infty, 0]$ in step $5a$ is sound, and when propagated, yields a conflict in step $6a$, so search returns unknown. An alternative run is $1, 2, 3b, 4b$. A decision $X_{\mathtt{in}}$:$[-\infty, -7]$ is sound, and propagation yields $X_{\mathtt{a}}$:$[-\infty, -7]$ and $X_{\mathtt{ex}}$:$\bot$, so search returns divergent. $\lhd$

Algorithm 3: Propagate()

$asg \leftarrow [tr]$
foreach $S \in scc(Inf_G)$ **do**
  $asg' \leftarrow \mathsf{Reach}(S, asg)$
  foreach $X_v{:}m \in mdiff(asg', asg)$ **do**
    $tr \leftarrow tr{\cdot}(X_v{:}m)$
foreach $\psi \in \Psi$ **do**
  $tr \leftarrow gunit(tr, \psi)$

Algorithm 4: Analyze()

$dc \leftarrow \{j \mapsto \top \mid 0 \leq j \leq |tr|\}$
$dc[|tr|] \leftarrow \{|tr| \mapsto [tr](X_{\mathtt{ex}}) \rightarrow 0\}$
$i \leftarrow |tr|$
repeat
  if $dc[i] = \top$ *or* $exp[i] = nil$ **then continue**
  $rk \leftarrow \mathsf{Term}(exp[i], dc[i])$
  $dc[i] \leftarrow \top$
  $i \leftarrow i - 1$
  $\mathsf{Update}(dc, tr, rk)$
until *Unique Implication Point*
return $[dc]$

Propagate(). Alg. **??** calls an abstract interpreter and stores the results in the trail in a form amenable to conflict analysis and learning. The notion of *meet-difference* makes explicit the incremental change between two calls to the abstract interpreter. Formally, the *meet-difference* of $a, b \in A$ $mdiff(a, b) = mdc(a) \setminus mdc(b)$. The meet-difference of two abstract assignments is the pointwise lift $mdiff(asg, asg') = \{X_{\mathtt{v}}{:}m \mid m \in mdiff(asg(X_{\mathtt{v}}), asg'(X_{\mathtt{v}})), X_{\mathtt{v}} \in SVar\}$.

In a transition constraint $\psi \mathrel{\hat{=}} \forall i.\forall j.X_{\mathtt{v}}(j) \wedge Suc(i, j) \Rightarrow \ldots$, we write $sink(\psi)$ for $X_{\mathtt{v}}$. A *strongly connected component (*SCC*) of* Inf_G is a set of transition constraints T such that the set of locations $\{v \mid \psi \in T, X_{\mathtt{v}} = sink(\psi)\}$ is an SCC of G. The set of SCCs of Inf_G is $scc(Inf_G)$. Propagate() calls a standard abstract interpreter on each SCC and uses a meet-difference calculation to extend the trail with new information. Propagate() also applies a generalized unit rule *gunit*, explained in § **??**. Propagation is sound in the sense that it does not eliminate models of the constraints involved.

Lemma 1. *If* (τ, σ) *satisfies* Inf_G *and* Ψ *and is in* $conc([tr])$*, it is also in* $conc([tr])$ *after invoking* Propagate().

Decisions. The abstract assignment computed by (the abstract interpreter used by) Propagate() can be refined using decisions. Boolean decisions make variables true or false and first-order decisions use values [**?**,**?**] but our decisions, like those in [**?**], use abstract domain elements.

A *decision* is an element $X{:}m$ that can be on a trail. A decision is *sound* if $conc(X{:}m) \cup conc(X{:}\overline{m}) = Struct$. That is, considering the structures in m and $\overline{m}$ amounts to considering all possible structures.

Example 4. Recall the unsound decision $X_{\mathtt{in}}{:}[9, \infty]$ from Ex. **??**. The structure (τ, σ) with $\tau = 9, 9, 8, 8, \ldots$ and σ partitioning $X_{\mathtt{in}}$ and $X_{\mathtt{a}}$ into even and odd values is not in $conc(X_{\mathtt{in}}{:}[9, \infty])$ as x cannot be 8 at `in`. Similarly, it is not in $conc(X_{\mathtt{in}}{:}[-\infty, 8])$ so $conc(X_{\mathtt{in}}{:}[9, \infty]) \cup conc(X_{\mathtt{in}}{:}[-\infty, 8]) \neq Struct$. ◁

The unsoundness arises because pointwise lifting does not preserve concrete complements. Though $\overline{m}$ is the concrete complement of m in A, $[X_v{:}\overline{m}]$ need not

be the concrete complement of $[X_v{:}m]$ in Asg_A. Unsound decisions can be extended by propagation to a post-fixed point to cover all structures. All decisions on variables X_v in singleton SCCs with no self-loops are sound.

A *decision rule* $\mathsf{dec}(Inf_G, \Psi, tr)$ returns an abstract domain element d such that $[tr{\cdot}(X_v{:}d)] \sqsubseteq [tr]$. The decision rule *makes progress* if this order is strict. Unlike in SAT the decision rule can cause divergence of CDCT because an infinite series of decisions like $[0,\infty], [1,\infty], \ldots$ may not change the result of propagation.

5 Conflict Analysis

Unlike SAT and SMT solvers, which generate definite conflicts, Search() generates potential conflicts. We apply backwards abstract interpretation with ranking functions to extract definite conflicts, and use widening to generalize them.

Ranking Function Domains. Due to space limitations, we only briefly recall the concrete domain of ranking functions, which provides the intuition for conflict analysis, and discuss the abstract domain informally. See [**?**,**?**] for details.

We write $f : A \rightharpoonup B$ for a partial function whose domain is $dom(f)$. A *ranking function* $f : State \rightharpoonup \mathbb{O}$ for a relation R is a map from states to ordinals satisfying that for all s in $dom(f)$ and (s,t) in R, t is in $dom(f)$ and $f(t) < f(s)$. A concrete domain for termination analysis $(Rank, \preccurlyeq, B)$ is a lattice of ranking functions with backwards transformers $B = \{bkw_c \mid c \in Cmd\}$ defined below. Informally $f \preccurlyeq g$ if f is defined on a state when g is and yields a lower rank: $f \preccurlyeq g \mathrel{\hat{=}} dom(f) \supseteq dom(g)$ and for all x in $dom(g)$, $f(x) < g(x)$. The transformer bkw_c maps a ranking function f to one defined on states with all their successors in $dom(f)$. Recall that Rel_c is the transition relation for a command c.

$$bkw_c(f) \mathrel{\hat{=}} \lambda s. \begin{cases} 0 & \text{if } Rel_c(s) = \emptyset \\ \sup\{f(r) \mid r \in Rel_c(s)\} + 1 & \text{if } Rel_c(s) \subseteq dom(f) \\ \text{undefined} & \text{otherwise} \end{cases}$$

A subset $P \subseteq A$ of a domain A is an *abstract partition* if $\{\gamma(a) \mid a \in P\}$ partitions $State$. Let $Fun \subseteq Rank$ be a lattice of functions, for example, affine functions.

A *piecewise defined ranking function* (PDRF) over Fun and A is a set $\rho \mathrel{\hat{=}} \{a_1 \mapsto f_1, \ldots, a_k \mapsto f_k\}$ such that $\{a_1, \ldots, a_k\}$ is an abstract partition, and each f_i is in Fun. The abstract domain of PDRFs $(aRank, \preccurlyeq, Abd)$ is a lattice $aRank$ with *abduction transformers* Abd. The concretization $\gamma^r : aRank \to Rank$ of a ρ as above maps states to ranking functions: $\gamma^r(\rho) \mathrel{\hat{=}} \{s \mapsto f_i \mid s \in \gamma(a_i)\}$. The order and lattice operations are defined in terms of partition refinement and unification [**?**]. To compare ρ_1 and ρ_2, we consider the coarsest abstract partition that refines the abstract partitions of both and compare the ranking functions in each block pointwise.

Conflict analysis starts with a precondition for termination and finds a weaker precondition for termination, hence performs abduction. The abduction transformers satisfy the soundness condition: $\gamma^r(abd_c(\rho)) \preccurlyeq bkw_c(\gamma^r(\rho))$, which states that the termination bounds obtained with PDRFs are weaker than those that

could be obtained in the concrete domain. A sound abduction transformer is underapproximating. A *ranking assignment* $rk : SVar \to aRank$ associates a PDRF with each second-order variable. Ranking assignments form a lattice with point-wise meet and join and have a special order $\leqslant$ for fixed point checks [**?**]. To exchange information between Analyze() and Search() we extract a meet-irreducible representation of the domains of PDRFs. The *meet-projection* of a PDRF $\rho \mathrel{\hat{=}} \{a_i \mapsto f_i\}$ is the set of sets of meet-irreducibles $mpr(\rho) \mathrel{\hat{=}} \{mdc(a_i)\}$ and provides a DNF-like representation of the abstract partition in ρ.

Analyze(). Alg. **??** uses an array dc to construct and generalize a definite conflict. Each $dc[i]$ represents termination conditions for states in the trail. Executions from states at ex terminate immediately so the last element of dc is $\{X_{\mathtt{ex}} \mapsto \{[tr](X_{\mathtt{ex}}) \mapsto 0\}\}$ and all other elements are $\top$. The conflict analysis loop walks backwards through the trail and extends $dc[i]$. Forward propagation through the SCC $exp[i]$ added $tr[i]$ to the trail, so $dc[i]$ is propagated backwards through $exp[i]$ to generalize the conflict to a ranking assignment rk. New PDRFs are added to dc by the procedure Update(). Specifically, for each X_v modified by Term(), and $m \in mpr(rk(X_v))$, Update() finds trail indices with $tr[j] \sqsubseteq X_v{:}m$ and sets $dc[j]$ to the appropriate PDRF. Analyze() continues until a *unique implication point* is reached, which is typically a dominator in the CFG at which a decision was made. Analyze() returns $[dc]$, a representation of the PDRFs in dc.

Learn() ***and the Generalized Unit Rule.*** Information computed by Search() is communicated to Analyze() using the trail, while information from Analyze() is represented within Search() by a blocking constraint and is incorporated in search using generalized unit rule. We describe these very briefly.

A set $C = \{X_1{:}m_1, \ldots, X_k{:}m_k\}$ of elements can be complemented element-wise to obtain $\overline{C} = \{X_1{:}\overline{m}_1, \ldots, X_k{:}\overline{m}_k\}$. If C is viewed as a conjunction of literals representing a conflict, $\overline{C}$ is a clause the procedure can learn. Learn() applies meet-projection to a PDRF and complements this projection to obtain a *blocking constraint*. In practice, we simplify the partitions of the PDRF to avoid an explosion of blocking constraints, analogous to subsumption in SAT.

The generalized unit rule [**?**] extends a trail using a blocking constraint. Assume that Ψ has the form $\{X_0{:}m_0, \ldots, X_k{:}m_k\}$. The trail $gunit(tr, \Psi)$ is $tr \cdot (X_k{:}m_k)$ if $[tr](X_i) \sqcap m_i = \bot$ for $0 \leq i < k$ and is tr otherwise. The generalized unit rule refines a trail in the sense that $[gunit(tr, \Psi)] \sqsubseteq [tr]$. If tr is inconsistent with Ψ, $[tr]$ will represent $\bot$. Having presented all components of the procedure, we now investigate how it works in practice.

6 Implementation

We have incorporated CDCT in our prototype static analyzer FuncTion (`http://www.di.ens.fr/~urban/FuncTion.html`), which is based on piecewise-defined ranking functions [**?**]. A version without CDCT [**?**] participated in the *4th International Competition on Software Verification (SV-COMP 2015)*.

FuncTion+CDCT accepts (non-deterministic) programs in a C-like syntax. It is implemented in OCaml and uses the APRON library [**?**]. The pieces of a

(a)

	Tot	Time	Timeouts
FuncTion+cdct	200	1.5s	15
AProVE [?]	256	15.9s	24
FuncTion [?]	175	0.7s	5
HIPTnT+ [?]	246	1.2s	4
Ultimate [?]	226	15.3s	35

(b)

	FuncTion+cdct			
	■	▲	×	○
AProVE [?]	15	71	185	17
FuncTion [?]	25	0	175	88
HIPTnT+ [?]	22	68	178	20
Ultimate [?]	41	67	159	21

Fig. 3: Overview of the experimental evaluation.

pdrf can be represented with intervals, octagons or convex polyhedra, and ranking functions within the pieces are represented by affine functions. The precision of the analysis can also be controlled by adjusting the widening delay.

Experimental Evaluation. We evaluated our tool against 288 terminating C programs from the termination category of *SV-COMP 2015*. In particular, we compared FuncTion+cdct with other tools from the termination category of *SV-COMP 2015*: AProVE [?], FuncTion without cdct [?], HIPTnT+ [?], and Ultimate Automizer [?]. The experiments were performed on a system with a 1.30GHz 64-bit Dual-Core CPU (Intel i5-4250U) and 4GB of RAM. For the other tools, since we did not have access to their competition version, we used the *SV-COMP 2015* results obtained on more powerful systems with a 3.40GHz 64-bit Quad-Core CPU (Intel i7-4770) and 33GB of RAM.

Fig. **??** summarizes our evaluation. The first column is the number of programs each tool could prove terminating. The second column reports the average running time in seconds, and the last column reports the number of time outs, which was set to 180 seconds. In Fig. **??**, the first column (■) lists the number of programs that FuncTion+cdct proved terminating and the tool could not, the second column (▲) reports the number of programs that the tool proved terminating and FuncTion+cdct could not, and the last two columns report the number of programs that the tool and FuncTion+cdct were both able (×) or unable (○) to prove terminating. The same symbols are used in Fig. **??**.

Fig. **??** shows that cdct causes a 9% improvement in FuncTion+cdct compared to FuncTion without cdct. The increase in runtime is not evenly distributed, and about 2% of the test cases require more than 20 seconds to be analyzed by FuncTion+cdct (cf. Fig. **??**). In these cases the decision heuristics do not quickly isolate sets of states on which the abstract interpreter makes progress. Fig. **??** shows that, as expected, FuncTion without cdct terminates with an unknown result earlier. Fig. **??** and Fig. **??** show that though AProVE and Ultimate Automizer were run on more powerful machines, FuncTion+cdct is generally faster but proves termination of respectively 19% and 9% fewer programs (cf. Fig. **??**). HIPTnT+ proves termination of 16% more programs than FuncTion+cdct (cf. Fig. **??**), but FuncTion+cdct proves termination of 52% of the program that HIPTnT+ is not able to prove terminating (8% of the total test cases, cf. Fig. **??**). When comparing with FuncTion

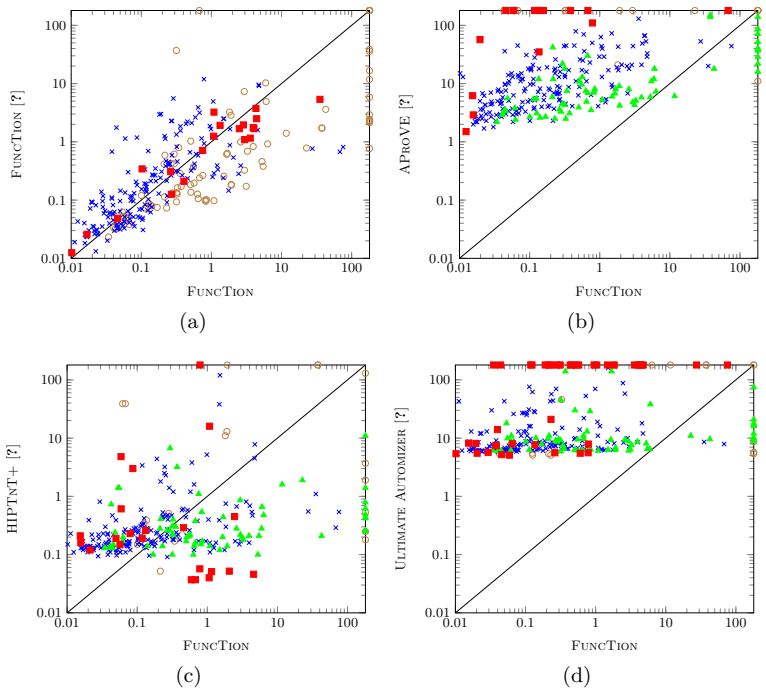


Fig. 4: Detailed comparison of FuncTion against its previous version [?] (a), AProVE [?] (b), HIPTnT+ [?] (c), and Ultimate Automizer [?] (d).

without cdct [?], we observed a 2x speedup in the *SV-COMP 2015* machines, so the runtime comparison of FuncTion+cdct and HIPTnT+ is inconclusive. Finally, thanks to the support for piecewise-defined ranking functions, 1% of the programs could be proved terminating only by FuncTion+cdct (2.7% by AProVE, 1% by HIPTnT+, and 1.7% by Ultimate Automizer). No tool could prove termination for 0.7% of the programs.

7 Related Work and Conclusion

Büchi's work relating automata and logic [?] is the basis for automata-based verification and synthesis. We depart from most work in this tradition in two ways. One is the use of sequences of first-order structures as in first-order temporal logics [?] and the other is to go from a graph-based representation to a formula, which is opposite of the translation used in automata-theoretic approaches. The use of s1s for pointer analysis [?], and termination [?] is restricted to decidable cases, as is [?]. Program analysis questions have been formulated with set-constraints [?] and second-order Horn clauses [?], but solutions to these formulae

are typically invariants and ranking functions, not errors, and the methods used to solve them differ from CDCT.

A key intuition behind our work is to lift algorithmic ideas from SAT solvers to program analysis. The same intuition underlies SMPP [**?**], which lifts DPLL(T) to programs, ACDCL [**?**,**?**], which lifts CDCL to lattices, the lifting of Stålmarck's method [**?**], and lazy annotation, which uses interpolants for learning [**?**]. The idea of guiding an abstract interpreter away from certain regions appears in DAGGER [**?**] and VINTA [**?**], from which CDCT differs in the use of a trail in search and a unit rule in learning. Our generalized unit rule is from ACDCL, but the use of S1S(T), potential conflicts and the combination with PDRFs is all new. The widening used in CDCT preserves a termination guarantee and we believe that algorithms for generating small interpolants [**?**] can help design better widening operators.

Finally, termination analysis is a thriving area with more approaches than we can discuss. A fundamental problem is the efficient discovery of disjunctions of ranking functions [**?**]. We use backward analysis, as in [**?**,**?**], and our combination of conditional termination [**?**] with non-termination [**?**,**?**] is crucial. The approach of [**?**] is similar ours with a different refutation step and information exchange mechanism. At a high level, CDCT is the dual of [**?**], which underapproximates non-terminating executions and overapproximates terminating ones, while we overapproximate non-termination and underapproximate termination. We believe CDCT can be extended to transition-based approaches [**?**], but the challenge is to develop search and learning.

References

1. A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, November 1999.
2. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig interpretation. In *SAS*, pages 300–316, 2012.
3. A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *CAV*, pages 313–329, 2013.
4. M. Brain, V. D'silva, A. Griggio, L. Haller, and D. Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in Systems Design*, 45(2):213–245, Oct. 2014.
5. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science*, pages 1–11. 1960.
6. B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving Conditional Termination. In *CAV*, pages 328–340, 2008.
7. S. Cotton. Natural domain SMT: A preliminary assessment. In *FORMATS*, pages 77–91, 2010.
8. P. Cousot and R. Cousot. An Abstract Interpretation Framework for Termination. In *POPL*, pages 245–258, 2012.
9. C. David, D. Kroening, and M. Lewis. Unrestricted termination and non-termination arguments for bit-vector programs. In *ESOP*, pages 183–204, 2015.
10. V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. In *POPL*, pages 143–154, 2013.

11. V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric bounds analysis with conflict-driven learning. In *TACAS*, pages 48–63, 2012.
12. P. Ganty and S. Genaim. Proving Termination Starting from the End. In *CAV*, pages 397–412, 2013.
13. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
14. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS*, pages 443–458, 2008.
15. A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *POPL*, pages 147–158, 2008.
16. W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani. Alternation for termination. In *SAS*, pages 304–319, 2010.
17. W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL*, pages 71–82, 2010.
18. M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. Ultimate Automizer with Array Interpolation (Competition Contribution). In *TACAS*, 2015.
19. I. M. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable and undecidable fragments of first-order branching temporal logics. In *LICS*, pages 393–402, 2002.
20. B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, pages 661–667, 2009.
21. D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving Non-Termination Using Max-Smt. In *CAV*, pages 779–796, 2014.
22. T.-C. Le, S. Qin, and W.-N. Chin. Termination and Non-Termination Specification Inference. In *PLDI*, 2015.
23. K. L. McMillan. Lazy annotation for program testing and verification. In *CAV*, pages 104–118, 2010.
24. K. L. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to richer logics. In *CAV*, pages 462–476, 2009.
25. F. Mesnard and É. Payet. A second-order formulation of non-termination. *CoRR*, 2014.
26. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.
27. A. Podelski and A. Rybalchenko. Transition Invariants. In *LICS*, pages 32–41, 2004.
28. A. Podelski and A. Rybalchenko. Transition Invariants and Transition Predicate Abstraction for Program Termination. In *TACAS*, pages 3–10, 2011.
29. T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. AProVE: Termination and Memory Safety of C Programs (Competition Contribution). In *TACAS*, 2015.
30. A. Thakur and T. Reps. A generalization of Stålmarck's method. In *SAS*, 2012.
31. C. Urban. The Abstract Domain of Segmented Ranking Functions. In *SAS*, pages 43–62, 2013.
32. C. Urban. FuncTion: An Abstract Domain Functor for Termination (Competition Contribution). In *TACAS*, 2015.