Model Checking Recursive Programs with Exact Predicate Abstraction

Arie Gurfinkel[1], Ou Wei[2], and Marsha Chechik[2]

[1] Software Engineering Institute, Carnegie Mellon University
[2] Department of Computer Science, University of Toronto

Abstract. We propose an approach for analyzing non-termination and reachability properties of recursive programs using a combination of over- and under-approximating abstractions. First, we define a new concrete program semantics, *mixed*, that combines both natural and operational semantics, and use it to design an on-the-fly symbolic algorithm. Second, we combine this algorithm with abstraction by following classical fixpoint abstraction techniques. This makes our approach parametrized by different approximating semantics of predicate abstraction and enables a uniform solution for over- and under-approximating semantics. The algorithm is implemented in YASM, and we show that it can establish non-termination of non-trivial C programs completely automatically.

1 Introduction

Automated predicate abstraction is one of the key techniques for extending finite-state model-checking to software. It combines automated construction of a finite abstract model with automated analysis by model-checking and iterative abstraction refinement. Traditionally, predicate abstraction is an over-approximation of a program and thus is biased towards establishing correctness of safety properties. To exploit the bug detection ability of model-checkers and to extend the scope of abstract model-checkers to richer properties, recent research has proposed abstract analysis that combines both over- and under-approximations [9, 15, 25, 26, 4, 18, 17]. Although such a combination, which we call *exact-approximation*, has been shown to be effective in practice [17, 19], until now this line of research has focused exclusively on analyzing non-recursive programs. In this paper, we propose a novel approach to extend such over- and under-approximating analyses to *recursive* programs. Our approach has been implemented in a software model-checker YASM. We illustrate it on non-termination and reachability analysis of several C programs, including the benchmarks from BEBOP [6], VERA [1], and MOPED [14, 8], the `Ack` program from [10] and a buggy version of `Quicksort` from [14]. To our knowledge, this is the first time that *non-termination* of such C programs was established completely automatically.

As a motivation, we review an over-approximation-based approach for model-checking of non-recursive programs and its limitations. Assume we want to check whether the `ERROR` label is reachable in the C program $\texttt{EX}_0$ shown in Figure 1(a). This safety property is expressed in CTL as $\varphi : AG\ (pc \neq \texttt{ERROR})$. An over-approximating abstraction $\alpha(\texttt{EX}_0)$ of $\texttt{EX}_0$ using the predicate $p : x > 0$ is shown in Figure 1(b), where '$*$' is interpreted as a non-deterministic choice. $\alpha(\texttt{EX}_0)$ is a finite *boolean* model

Cha et al. (Eds.): ATVA 2008, LNCS 5311, pp. 95–110, 2008.

(a)

```
1. x=read(); y=read();
2. if(x>0){
3.   while(x>0) {
4.     x=x+1;
5.     if(x<=0) ERROR;}
6. } else
7.     while(y>0) y=y-1;
8. END;
```

(b)

```
1. p = *;
2. if(p){
3.   while(p) {
4.   p = p?true:*;
5.     if(!p) ERROR;}
6. } else
7.     while(*) p = p;
8. END;
```

Fig. 1. (a) A program $\texttt{EX}_0$, and (b) its over-approximation $\alpha(\texttt{EX}_0)$ using predicate $p : x > 0$

which over-approximates the original program: it contains all feasible and some infeasible (or spurious) executions. For example, $\alpha(\texttt{EX}_0)$ has an execution which gets stuck in the `while(*)` loop on line 7, but $\texttt{EX}_0$ does not have the corresponding execution. Thus, if a universal temporal property, i.e., in the one expressed in ACTL, holds in $\alpha(\texttt{EX}_0)$, it also holds in $\texttt{EX}_0$. For example, our property φ is satisfied by $\alpha(\texttt{EX}_0)$, which means `ERROR` is unreachable in $\texttt{EX}_0$. However, when a property is falsified by $\alpha(\texttt{EX}_0)$, the result cannot be trusted since it may be caused by a spurious behavior. For example, consider checking whether $\texttt{EX}_0$ always terminates, i.e., whether it satisfies $\psi : AF\ (pc = \texttt{END})$. ψ is falsified on our abstraction, but this result cannot be trusted due to the infeasible non-terminating execution around the `while(*)` loop on line 7.

The falsification (or refutation) ability of predicate abstraction can be dramatically improved by using an *under*-approximating abstraction, where each abstract behavior is simulated by some concrete one. In this case, if a bug (or an execution) is present in the abstract model, it *must* exist in the concrete program. For example, the predicate p *must* always be *true* in the `while(p)` loop at line 3 (assuming `int` is interpreted as mathematical integers). Thus, an under-approximation based on predicate p is sufficient to establish that $\texttt{EX}_0$ is non-terminating.

There has been a considerable amount of research exploring abstract analysis based on a combination of over- and under-approximating abstractions, e.g., [9, 15, 25, 26, 4, 18, 17]. Compared with an analysis based on over-approximation alone, there are two main differences:

1. Such a combination requires a non-boolean abstract model that can represent both over- and under-approximations at the same time. Examples of such models are *Modal Transition Systems* [21] (equivalently, 3-valued Kripke structures [9]) and *Mixed Transition Systems* [13] (equivalently, 4-valued Kripke structures [18]). These models use two types of transitions: *may* for over-approximation, and *must* for under-approximation.
2. It requires new model-checking algorithms for these models, such that a formula is evaluated to either *true* or *false*, which are trusted, or to *unknown*, which indicates that the abstraction is not precise enough for a conclusive analysis.

Although both theoretical and practical combinations of exact-approximation with automated CounterExample Guided Abstraction Refinement have been explored, they are all limited to analyzing non-recursive programs.

One way to extend such analysis to recursive programs is to continue to mirror the traditional approach, i.e., (a) extend push-down systems to support combined over- and under-approximations, and (b) develop analysis algorithms for this new modeling

formalism. While this approach seems natural, we are not aware of any existing work along this line.

In this paper, we propose an alternative solution to this problem. Our approach does not require the development of new specialized types of push-down systems, nor new specialized analysis algorithms. The key to our approach is to separate the analysis of recursive programs from abstraction of the data domain. We accomplish this by introducing a new concrete program semantics, which we call *mixed*, and using it to derive efficient symbolic algorithms for the analysis of non-termination and reachability properties of finite recursive programs. These algorithms share many insights with techniques in related work [8, 6, 1], i.e., they are functional [24] in terms of interprocedural analysis, and apply only to *stack-independent* properties. The novelty of our approach is the formalization of the algorithms as equational systems, and the parametrization of the algorithms by data abstractions. This makes it possible to share the same analysis algorithms for over-, under-, and exact-approximations! In particular, we demonstrate that in combination with exact-approximation [17], our abstract analysis supports both verification and refutation.

The rest of this paper is organized as follows. We present preliminaries and fix our notation in Sec. 2. We present a simple programming language PL and its natural, and operational semantics in Sec. 3. In Sec. 4, we introduce mixed semantics and derive symbolic on-the-fly algorithms for analyzing recursive programs with finite data domain for reachability and non-termination. In Sec. 5, we parametrize the algorithms of Sec. 4 by abstraction for handling programs with infinite data domain. Experiments are reported in Sec. 6, and we conclude in Sec. 7. Additional illustrations are given in the Appendix.

2 Preliminaries

Valuation and Relations. A *valuation* σ on a set of typed variables V is a function that maps each variable x in V to a value $\sigma(x)$ in its domain. We assume that valuations extend to expressions in the obvious way. The domain of σ is called a *valuation type* and is denoted by $\tau(\sigma)$. For example, if $\sigma = \{x \mapsto 5, y \mapsto 10\}$ then $\tau(\sigma) = \{x, y\}$. The projection of σ on a subset $U \subseteq V$ is denoted by $\sigma|_U$.

The set of all valuations over V is denoted by $\Sigma_V \triangleq \{\sigma \mid \tau(\sigma) = V\}$. Note that $\Sigma_\emptyset$ is well-defined and consists of the unique empty valuation. A relation r on two sets of variables U and V is a subset of $\Sigma_U \times \Sigma_V$. The *relational type* of r is $U \to V$, denoted by $\tau(r)$. For example, the type of $x' = y$ is from y to x, that is, $\tau(x' = y) = \{y\} \to \{x\}$. In this paper, we use several simple relations: $true$ is the **true** relation, id is the identity relation (e.g., $id(x) \triangleq x' = x$), $decl$ is a relation for variable declaration, and *kill* — for variable elimination. Formally, they are defined as follows, with the format *name* '$\triangleq$' *expression* ':' *type*:

$$\begin{array}{llll} \mathit{true}(U \to V) & \triangleq \Sigma_U \times \Sigma_V : U \to V & \mathit{decl}(V) & \triangleq true(\emptyset \to V) : \emptyset \to V \\ \mathit{kill}(V) & \triangleq true(V \to \emptyset) : V \to \emptyset & \mathit{id}(V) & \triangleq \{(\sigma, \sigma') \in \Sigma_V \times \Sigma_V \mid \sigma = \sigma'\} : V \to V \end{array}$$

Operations on relations are defined in Table 1, where $\vee$, $\circ$ and $\times$ are *asynchronous*, *sequential* and *parallel* composition, respectively, *assume* is a restriction of identity

Table 1. Relational operations

Operation	Assumption	Definition	Type
$r_1 \vee r_2$	$\tau(r_1) = \tau(r_2)$	$\lambda a, a' \cdot r_1(a, a') \vee r_2(a, a')$	$\tau(r_1)$
$r_1 \circ r_2$	$\tau(r_1) = U \to V$ $\wedge\ \tau(r_2) = V \to W$	$\lambda a, a' \cdot \vee_{a''} (r_1(a, a'') \wedge r_2(a'', a'))$	$U \to W$
$r_1 \times r_2$	$\tau(r_1) = U \to V_1$ $\wedge\ \tau(r_2) = U \to V_2$ $\wedge\ V_1 \cap V_2 = \emptyset$	$\lambda a, a' \cdot r_1(a, a'\|_{V_1}) \wedge r_2(a, a'\|_{V_2})$	$U \to (V_1 \cup V_2)$
assume(Q)		$\lambda a, a' \cdot Q(a) \wedge id(\tau(Q))(a, a')$	$\tau(Q) \to \tau(Q)$
$[W]r$	$\tau(r) = U \to V$	$\lambda a, a' \cdot r(a\|_U, a')$	$(U \cup W) \to V$
$(W \to Z)r$	$\tau(r) = U \to V \wedge U \subseteq W \wedge (Z \setminus V) \subseteq W$	$([W]r) \times ([W](id(Z \setminus V)))$	$W \to Z$

relation to a set Q of valuations, $[\cdot]$ is *variable introduction*, and $(\cdot \to \cdot)$ is *scope extension*. Note that $\times$ combines the outputs of two relations, and $[\cdot]$ extends the source of a relation with new variables. Together these operators allow constructing complex relations from simple ones. For example, $[\{x, y\}](x' = y) \times [\{x, y\}](y' = x)$ is the relation $(x' = y) \wedge (y' = x)$ with the type $\{x, y\} \to \{x, y\}$. Directly composing $x' = y$ and $y' = x$ without variable introduction, i.e., $(x' = y) \times (y' = x)$, is invalid because $\tau(x' = y) = \{y\} \to \{x\}$ and $\tau(y' = x) = \{x\} \to \{y\}$ have different source types. Scope extension extends a relation by combining it with the identity on new variables. For example, $(\{x, y\} \to \{x, y\})(x' = x + 1)$ is $(x' = x + 1) \wedge (y' = y)$. The assumptions for scope extension ensure that any new variable introduced in the destination of r must also be available in the source. For example, the extension $(\{x, y\} \to \{x, z\})(x' = x + 1)$ is not allowed since z is not available in the source of the relation.

For a relation r with a type $U \to V$, we define the *pre-image* of $Q \subseteq \Sigma_V$ w.r.t. r, $pre[r] : \mathbf{2}^{\Sigma_V} \to \mathbf{2}^{\Sigma_U}$, as

$$pre[r](Q) \triangleq \lambda a \cdot \vee_{a'} (r(a, a') \wedge Q(a'))$$

Reachability and Non-termination. A Kripke structure $K = \langle \mathcal{S}, \mathcal{R} \rangle$ is a transition system, where $\mathcal{S}$ is a set of states and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation.

Let p be an atomic proposition, and $\mathcal{S}_p \triangleq \{s \in \mathcal{S} \mid s \models p\}$ be the set of states satisfying p. A *reachability* property (*EF* p in CTL) is true in a state s if there exists a path from s to a state in $\mathcal{S}_p$. A *non-termination* property (*EG* p in CTL) is true in a state s if there exists an infinite path starting at s and contained in $\mathcal{S}_p$.

The set RS of all states satisfying *EF* p is the least solution to equation reach, and the set NT of all states satisfying *EG* p is the greatest solution to equation non-term:

$$RS = \mathcal{S}_p \cup pre[\mathcal{R}](RS) \quad \text{(reach)} \qquad NT = pre[\mathcal{R} \cap \mathcal{S}_p](NT) \quad \text{(non-term)}$$

3 Programming Language and Semantics

We use a simple imperative programming language PL which allows non-determinism and recursive function calls. We assume that (a) functions have a set of call-by-value formal parameters and a set of return variables; (b) each variable has a unique name and explicit scope; (c) there are no global variables (they can be simulated by local variables); and (d) a type expression is associated with each statement and explicitly defines the pre- and post-variables of the statement.

(a)

```
1:  f1() {
2:  en1: skip;
3:  var x,y;
4:  x,y := 3,0;
5:  x := f2(x);
6:  skip;
7:  while (x==2 && y<=0) {
8:    y := f2(y);
9:  }
10: kill x,y;
11: ex1: }

12: f2(z):z {
13: en2: skip;
14: while (z < 0) {
15:   z := z+1;
16: }
17: z := z-1;
18: ex2: }
```

(b)

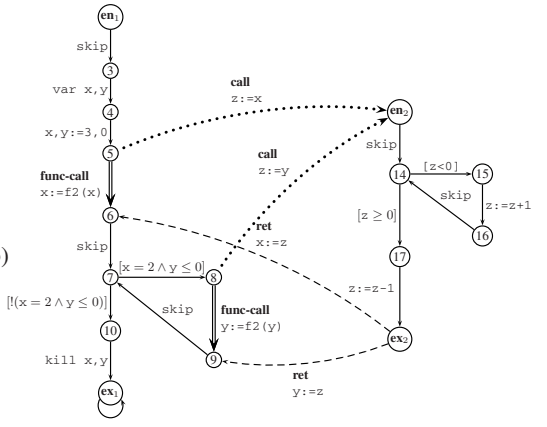

Fig. 2. (a) A program $\texttt{EX}_1$ and (b) its ICFG

Syntax. Let *var* denote variables, *func* function identifiers, *e* expressions, and *T* valuation types. The syntax of PL is defined as follows:

$$
\begin{array}{lll}
Atomic & ::= & \textbf{skip} \mid var^+ := e^+ \mid \textbf{assume}(e) \mid \textbf{var}\ var^+ \mid \textbf{kill}\ var^+ \mid (T \to T)Atomic \\
Stmt & ::= & Atomic \mid Stmt\,;Stmt \mid Stmt \,[\!]\, Stmt \mid \textbf{if}(e)\ \textbf{then}\ Stmt\ \textbf{else}\ Stmt \\
 & & \mid \textbf{while}(e)\ Stmt \mid var^+ := func(var^+) \mid (T \to T)Stmt \\
Fdef & ::= & func(var^+) : var^+\ Stmt \\
Prog & ::= & Fdef^+
\end{array}
$$

We use bold lower case letters to represent vectors, e.g., a statement $\mathbf{x} := \mathbf{e}$ means an assignment $x_1, \cdots, x_n := e_1, \cdots, e_n$. For a function f with declaration $f(p_1, \cdots, p_n) : r_1, \cdots, r_k$, $\mathbf{p}_f$ and $\mathbf{r}_f$ to denote the formal parameters and the return variables of f, respectively. $var(e)$ denotes the variables of e, and we assume that each program has a "main" function f_1, not called by other functions.

Base Semantics. Let Σ denote the set of all valuations in a PL program. With each *atomic* statement S, we associate *base semantics* that interprets the statement as a relation $[\![S]\!] \subseteq \Sigma \times \Sigma$ on valuations of program variables:

$$
[\![\textbf{skip}]\!] \triangleq id(\emptyset) \quad [\![\textbf{var}\ \mathbf{x}]\!] \triangleq decl[\mathbf{x}] \quad [\![\textbf{kill}\ \mathbf{x}]\!] \triangleq kill[\mathbf{x}] \quad [\![(U \to V)(S)]\!] \triangleq (U \to V)[\![S]\!]
$$
$$
[\![\mathbf{x} := \mathbf{e}]\!] \triangleq \{(\sigma, \sigma') \mid \tau(\sigma) = var(\mathbf{e}) \wedge \sigma' = [\mathbf{x}_i \mapsto \sigma(\mathbf{e}_i)]\}
$$
$$
[\![\textbf{assume}(e)]\!] \triangleq \{(\sigma, \sigma') \mid (\sigma, \sigma') \in id(var(e)) \wedge \sigma \models e\}
$$

Note that for the type cast statement $(U \to V)S$, we only consider those cases where the assumptions for the scope extension are satisfied.

Interprocedural Control Flow Graph. A PL program is represented by an *Interprocedural Control Flow Graph* (ICFG) [24]. An ICFG is a labeled graph $G = \langle Loc, Edge, \pi \rangle$, where Loc is a finite set of locations, $Edge \subseteq Loc \times Loc$ is a set of edges, and π labels each edge with a program statement. For example, the ICFG for the program $\texttt{EX}_1$ (see Fig. 2(a)) is shown in Fig. 2(b). In ICFGs, (a) each function has a unique *entry* (**en**) and *exit* (**ex**); (b) there is a self-loop at **ex** of f_1 to ensure existence of an infinite execution; (c) each function call (**func-call**) is: a **call** edge, where the values of actual

Table 2. The rules of operational and mixed semanics. U is the set of local variables in the scope of the function call; $[\![f]\!]$ is natural semantics, $\mathbf{p}_f$ are the formals, and $\mathbf{r}_f$ are the returns of f.

Statement $\pi(\langle k, l\rangle)$	**Operational Semantics** $r_{\langle k,l\rangle}$	**Mixed Semantics** $r^{\mathrm{m}}_{\langle k,l\rangle}$
func-call edge $(U \to U)\ \mathbf{x} := f(\mathbf{a})$	$\emptyset$	$(U \to U) \quad ([\![\mathbf{p}_f := \mathbf{a}]\!] \circ [\![f]\!] \circ [\![\mathbf{x} := \mathbf{r}_f]\!])$
call edge $S \equiv (U \to \mathbf{x})\ \mathbf{x} := \mathbf{e}$	$\Gamma_t = s \wedge (\sigma_k, \sigma_l) \in [\![S]\!]$	$[\![S]\!]$
ret edge $(U \to V)\ \mathbf{x} := \mathbf{r}$	let $(c, \sigma_c).\Gamma_c = \Gamma_s$ in $\Gamma_t = \Gamma_c \wedge l = ret(c)$ $\wedge\ \ \sigma_l = \sigma_c[\{\mathbf{x}_i \mapsto \sigma_k(\mathbf{r}_i)\}]$	$\emptyset$
Intraprocedural: S	$\Gamma_t = \Gamma_s \wedge (\sigma_k, \sigma_l) \in [\![S]\!]$	$[\![S]\!]$

parameters of the callee function are assigned to the formal parameters, a function body, and a **ret** edge, where the return values are assigned to the variables of the caller.

We assume that **call** and **ret** edges are uniquely determined by each other. For a **call** edge $(k, \mathbf{en})$ and the corresponding **ret** edge $(\mathbf{ex}, l)$, k is the call location, $\mathit{call}(l) \triangleq k$, and l is the return location, $\mathit{ret}(k) \triangleq l$.

Operational Semantics of a program $P = \langle Loc, Edge, \pi\rangle$ is a transition system $K = \langle \mathcal{S}, \mathcal{R}\rangle$. Each state in $\mathcal{S}$ is a stack of activation records where each record is of the form $\langle pc, \sigma\rangle$, where $pc \in Loc$ is a program counter, corresponding to a particular control location in P, and $\sigma \in \Sigma_{V(pc)}$ is the valuation for variables in the scope of pc (denoted by $V(pc)$). For a state $s = (k, \sigma_k).\Gamma$, (k, σ_k) is the *top* element of s, $top(s)$. For a pair of states $s = (k, \sigma_k).\Gamma_s$ and $t = (l, \sigma_l).\Gamma_t$, the transition relation $\mathcal{R}$ is defined as $\mathcal{R}(s, t) \triangleq \langle k, l\rangle \in Edge \wedge r_{\langle k,l\rangle}(s, t)$, where $r_{\langle k,l\rangle}$ is a deterministic (but not necessarily total) relation on $\mathcal{S}$ at the edge $\langle k, l\rangle$, as defined in the 2nd column of Table 2. An intraprocedural statement only modifies the top activation record, and a statement on a **call** or a **ret** edge pushes a new record or pops one, respectively. The transition relations on **func-call** edges are empty, i.e., these edges are removed.

Natural Semantics [22] (a.k.a. big-step) of a block of code S is a relation $[\![S]\!] \subseteq \Sigma \times \Sigma$ between the input and output of S: i.e., $(\sigma, \sigma') \in [\![S]\!]$ iff the execution of S on σ terminates and results in σ'. Natural semantics of a program $P \equiv f_1, \cdots, f_n$ is a set of relations, one per function, i.e., $[\![P]\!] = \langle [\![f_1]\!], \cdots, [\![f_n]\!]\rangle$.

The semantic rules for PL are defined compositionally on the syntax using the function $[\![\cdot]\!]_\rho$, where ρ is an environment mapping free fixpoint variables (used for loops and functions) to relations with an appropriate type. Natural semantics for atomic statements is the same as base semantics; the other cases are:

$$[\![S_1; S_2]\!]_\rho \triangleq [\![S_1]\!]_\rho \circ [\![S_2]\!]_\rho \qquad [\![\mu X \cdot \mathcal{S}(X)]\!]_\rho \triangleq \mathrm{lfp}\left(\lambda Z \cdot [\![\mathcal{S}(X)]\!]_{\rho\{X \mapsto Z\}}\right)$$
$$[\![S_1 \parallel S_2]\!]_\rho \triangleq [\![S_1]\!]_\rho \vee [\![S_2]\!]_\rho \qquad [\![\mathbf{x} := f(\mathbf{a})]\!]_\rho \triangleq [\![\mathbf{p}_f := \mathbf{a}; X_f; \mathbf{x} := \mathbf{r}_f]\!]_\rho$$
$$[\![X]\!]_\rho \triangleq \rho(X) \qquad [\![\mathbf{while}(e)\ S]\!]_\rho \triangleq [\![\mu X_w \cdot \mathbf{if}(e)\ \mathbf{then}\ (S; X_w)]\!]_\rho$$
$$[\![\mathbf{if}(e)\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2]\!]_\rho \triangleq [\![(\mathbf{assume}(e);\ S_1) \parallel (\mathbf{assume}(\neg e);\ S_2)]\!]_\rho$$

where **lfp** denotes for least fixpoint, $\tau(\rho(X_f)) = \mathbf{p}_f \to \mathbf{r}_f$ and $\tau(\rho(X_w)) = \tau([\![S]\!]_\rho)$. A program $P \equiv f_1, \cdots, f_n$ induces the system of equations

$$\rho(X_{f_i}) = [\![S_{f_i}]\!]_\rho \quad (1 \le i \le n) \tag{nat}$$

Natural semantics of P is the least fixpoint solution to this system, e.g., for the program $\mathtt{EX}_1$, natural semantics of f_2 is $(z > 0 \wedge z' = z - 1) \vee (z \le 0 \wedge z' = -1)$.

Theorem 1. *Let $P \equiv f_1, \cdots, f_n$ be a program and $K = \langle \mathcal{S}, \mathcal{R} \rangle$ be its operational semantics. A pair of activation records $(\langle k, \sigma_k \rangle, \langle l, \sigma_l \rangle)$ is in $[\![f_i]\!]$ iff there exists a path $s_0, \cdots, s_m$ in K such that $s_0 = \langle k, \sigma_k \rangle.\Gamma_0$ and $s_m = \langle l, \sigma_l \rangle.\Gamma_m$, such that $\Gamma_0 = \Gamma_m$, k and l are **en** and **ex** of f_i, respectively, and for all other $s_j = \langle p, \sigma_p \rangle.\Gamma_j$ either $\Gamma_j \neq \Gamma_0$ or p is not **ex** of f_i.*

4 Reachability and Non-termination Analysis

We now turn our attention to checking reachability and non-termination of recursive programs. Reachability can be reduced to finding the least fixpoint solution to the equation reach w.r.t. a transition system of operational semantics of a program (see Sec. 2). Similarly, non-termination corresponds to finding the greatest solution to the equation non-term. However, since operational semantics explicitly exposes a potentially unbounded call stack at each state, these equations must be solved over an infinite transition system (even when all program variables range over finite domains). Thus, the exact fixpoint solution may not be computable.

However, many program properties depend only on the top of the call stack: i.e., they are *stack-independent*. Analysis of such properties can be done using *stack-free* operational semantics in which everything except for the *top* activation record is abstracted away. In this section, we apply this idea to the analysis of *EF* p (reachability) and *EG* p (non-termination) properties, where p is a proposition that depends only on the top activation record. Without loss of generality, we further assume that p only depends on program locations, i.e., it is of the form $pc = x$.

4.1 Mixed Semantics

We start by defining a stack-free operational semantics, called mixed semantics, for PL programs which removes the call stack but preserves reachability and non-termination properties w.r.t. operational semantics of Sec. 3.

Intuitively, *mixed* semantics is a combination of operational and natural semantics, in which a program is executed as follows: an atomic statement is executed as usual; a function call `x := foo(y)` is executed as a *non-deterministic* choice between (a) executing `foo`, i.e., updating the top activation record according to natural semantics of `foo`, and (b) entering the body of `foo`, *and* forgetting all but the top activation record. Upon reaching the end of the main function, the execution enters a self-loop indicating the end of the program, and blocks at all other exit locations since it does not remember the origin of the call. For example, consider mixed execution of the program $\texttt{EX}_1$ starting from line 5 with $x = 3$ and $y = 0$. At this point, the execution can either (a) move to line 6 and decrease x by one according to natural semantics of f_2, or (b) move to `en2` (line 13), assign z to 3, and forget about x and y. Within f_2, the execution continues until it blocks at `ex2` (line 18) with $z = 2$.

Formally, mixed semantics of a program $P = \langle Loc, Edge, \pi \rangle$ is a Kripke structure $K^{\mathtt{m}} = \langle \mathcal{S}^{\mathtt{m}}, \mathcal{R}^{\mathtt{m}} \rangle$, where each state is a *single* activation record $\langle pc, \sigma \rangle$. For a pair of states $s = \langle k, \sigma_k \rangle$ and $t = \langle l, \sigma_l \rangle$, the transition relation is $\mathcal{R}^{\mathtt{m}}(s, t) \triangleq (\langle k, l \rangle \in Edge) \wedge r^{\mathtt{m}}_{\langle k,l \rangle}(\sigma_k, \sigma_l)$, where $r^{\mathtt{m}}_{\langle k,l \rangle}$ is a relation on valuations, as defined in

the 3rd column of Table 2. Note that $r_e^{\mathfrak{m}}$ for **ret** edges is empty, which is equivalent to removing those edges from the ICFG.

Mixed semantics preserves reachability and non-termination properties w.r.t. operational semantics. If an execution of a function f reaches a state s under the latter, then either s is a location within f, or it is inside some other function that f calls (directly or indirectly). The non-deterministic treatment of function calls in the former ensures that both of these cases are covered. Similarly, if there exists an infinite execution starting inside f, then either this execution lies within f, or f calls a function that does not return the control back to f. Again, both cases are captured by mixed semantics.

Theorem 2. *Let K and $K^{\mathfrak{m}}$ be operational and mixed semantics of a given program, respectively, and p be a propositional formula on control locations. Then, $(K \models EF\ p) \Leftrightarrow (K^{\mathfrak{m}} \models EF\ p)$ and $(K \models EG\ p) \Leftrightarrow (K^{\mathfrak{m}} \models EG\ p)$.*

When all variables of a given program P range over finite domains, mixed semantics of P is a finite Kripke structure. Theorem 2 implies the following analysis algorithm:

Step 1: compute natural semantics of P by solving equation nat;
Step 2: construct the structure $K^{\mathfrak{m}}$ following the rules of mixed semantics;
Step 3: solve equations reach or non-term on $K^{\mathfrak{m}}$ for reachability or non-termination, respectively.

While sound and complete, this algorithm is not efficient, since it relies on the (potentially unnecessary) computation of "full" natural semantics of all functions (for Step 2) and the construction of "full" mixed semantics before the analysis of the property can even begin. As a trivial example, consider checking $EF(pc = 5)$ on the program $\mathtt{EX}_1$. Since reachability of line 6 is irrelevant for this analysis, there is no need to construct the transition relation corresponding to **func-call** edge $\langle 5, 6\rangle$ and thus no need to compute natural semantics of f_2. Following this observation, in the rest of this section, we show that the three steps of the above algorithm can be combined into an *on-the-fly* algorithm that only computes the necessary parts of mixed and natural semantics.

4.2 On-the-Fly Reachability

Intuitively, the analysis of $EF\ p$ properties only needs a part of mixed semantics that is used for solving equation reach, and that, in turn, drives the computation of the necessary parts of natural semantics. To illustrate, consider checking $EF(pc = 8)$ on $\mathtt{EX}_1$. Natural semantics of f_2 is $[\![f_2]\!] \equiv (z > 0 \wedge z' = z - 1) \vee (z \leq 0 \wedge z' = -1)$. After a few iterations, the reachability algorithm computes a pre-condition $Q \equiv x = 2 \wedge y \leq 0$ for reaching line 8 from line 6. To determine a pre-condition for Q w.r.t. a function call `x:=f2(x)` at line 5, it needs to compute $pre[r^{\mathfrak{m}}_{\langle 5,6\rangle}](Q) = (x = 3 \wedge y \leq 0)$, where $r^{\mathfrak{m}}_{\langle 5,6\rangle} \equiv (y' = y) \wedge ((x > 0 \wedge x' = x - 1) \vee (x \leq 0 \wedge x' = -1))$ is the instantiation of $[\![f_2]\!]$ to the call site. However, instead of using the "full" version of $[\![f_2]\!]$, it is sufficient to compute a pre-condition that *assumes* Q as a post-condition, i.e., to restrict $r^{\mathfrak{m}}$ to $x' = 2$ (the relevant part of Q) yielding $\hat{r}^{\mathfrak{m}} \equiv y' = y \wedge x = 3 \wedge x' = 2$. $\hat{r}^{\mathfrak{m}}$ is an instantiation of $z = 3 \wedge z' = 2$ in the context of the call, obtained by (a) converting Q to a postcondition of f_2 by taking its pre-image over the **ret** edge (which eliminates y and renames x to z), and (b) restricting $[\![f_2]\!]$ to this post-condition: $[\![f_2]\!] \circ (assume(z = 2)) \equiv z = 3 \wedge z' = 2$.

We now formalize the above intuition. Recall that $V(k)$ stands for the set of variables in the scope of a location k. Let l be the return location of a function call to f_i, $Q \subseteq \Sigma_{V(l)}$ be a set of valuations at l, and the corresponding **ret** edge $\langle \mathbf{ex}_i, l\rangle$ be labeled with $\mathbf{x} := \mathbf{r}_{f_i}$. Then, function $prop(\langle \mathbf{ex}_i, l\rangle, Q) \triangleq pre\,[[\mathbf{x} := \mathbf{r}_{f_i};\ (\mathbf{x} \to V(l))\mathbf{var}\ (V(l) \setminus \mathbf{x})]]\,(Q)$ turns Q into a post-condition of f_i. Here, the pre-image w.r.t. **var** undeclares (or removes) all variables that are not changed by the call, and the pre-image w.r.t. **ret** edge turns the post-condition on $\mathbf{x}$ into the one on $\mathbf{r}_{f_i}$.

Let $RS : Loc \to \mathbf{2}^{\Sigma}$ map each location k to a subset of $\Sigma_{V(k)}$, and, as in Sec. 3, let ρ be the semantics environment, mapping each fixpoint variable to a relation of an appropriate type. The on-the-fly algorithm for reachability analysis is the equation system reach-otf:

$$RS(k) = \begin{cases} \Sigma_{V(k)} & \text{if } k \models p\ (k \in Loc) \\ RS(k) \cup \bigcup_{l \in succ(k)} pre[\hat{r}^{\mathrm{m}}_{\langle k,l\rangle}]\,(RS(l)) & \text{otherwise} \end{cases} \qquad \text{(reach-otf)}$$

$$\rho(X_{f_i}) = [\![S_{f_i}]\!]_\rho \circ assume\left(\bigcup_{l \in succ(\mathbf{ex}_i)} prop\,(\langle \mathbf{ex}_i, l\rangle, RS(l))\right) \quad (i \in [1..n])$$

where *succ* are the successors of a node in the ICFG, S_{f_i} is the body of f_i, and for $S \equiv \pi(\langle k, l\rangle)$, $\hat{r}^{\mathrm{m}}_{\langle k,l\rangle}$ is defined as:

$$\hat{r}^{\mathrm{m}}_{\langle k,l\rangle} = \begin{cases} (U \to U)\ ([\![\mathbf{p}_f := \mathbf{a}]\!] \circ \rho(X_{f_i}) \circ [\![\mathbf{x} := \mathbf{r}_f]\!]) & \text{if } S \equiv (U \to U)\ \mathbf{x} := f(\mathbf{a}) \\ [\![S]\!] & \text{otherwise} \end{cases}$$

This system is a combination of nat and reach, where *prop* is used to propagate the reachability information to the computation of natural semantics. Since reachability and natural semantics are both least solutions to equations reach and nat, respectively, we need the least solution to the above equation as well.

The following theorem shows that the analysis based on equation system reach-otf is sound, and computes only the necessary part of natural semantics.

Theorem 3. *Let $RS_\downarrow$ and $\rho_\downarrow$ be the least solutions to equation system* reach-otf. *Then,*

1. *$RS_\downarrow$ is the least solution to equation* reach *on K^{m};*
2. *$\forall i \in [1..n] \cdot \rho_\downarrow(X_{f_i}) \subseteq [\![f_i]\!]$;*
3. *for any ρ, if $RS_\downarrow$ is the least solution to the RS equations in* reach-otf *w.r.t. ρ, then $\forall i \in [1..n] \cdot \rho_\downarrow(X_{f_i}) \subseteq \rho(X_{f_i})$.*

Part 1 of Theorem 3 shows that $RS_\downarrow$ is the solution for the reachability analysis; part 2 – that $\rho_\downarrow$ is sound w.r.t. natural semantics of f_i; and part 3 – that $\rho_\downarrow$ only contains the information necessary for the analysis.

Since we need the least solution for both $RS(k)$ and $\rho(X_{f_i})$ equations, it can be obtained by any chaotic iteration [11] and thus is independent of the order of computation of RS and ρ. Interestingly, the algorithm derived from reach-otf is a pre-image-based variant of the post-image-based reachability algorithm of BEBOP [6], and is similar to the formalization of backward analysis with $\mathbf{wp}$ described in [3].

4.3 On-the-Fly Non-termination

The derivation of the on-the-fly algorithm for the analysis of non-termination, nt-otf, proceeds similarly, and is a combination of systems nat and non-term:

$$NT(k) = \begin{cases} \emptyset & \text{if } k \not\models p \ (k \in Loc) \\ \bigcup_{l \in succ(k)} pre[\hat{r}^{\mathtt{m}}_{\langle k,l\rangle}](NT(l)) & \text{otherwise} \end{cases} \qquad \text{(nt-otf)}$$

$$\rho(X_{f_i}) = [\![S_{f_i}]\!]_\rho \circ assume\left(\bigcup_{l \in succ(\mathbf{ex}_i)} prop\left(\langle \mathbf{ex}_i, l\rangle, NT(l)\right)\right) \quad (i \in [1..n])$$

where $NT : Loc \to \mathbf{2}^{\Sigma}$ maps each location k to a subset of $\Sigma_{V(k)}$, and *succ*, S_{f_i} and $\hat{r}^{\mathtt{m}}$ are the same as those in reach-otf. Since non-termination requires the greatest solution to non-term, and natural semantics – the least solution to nat, in nt-otf, we need the greatest solution to $NT(k)$, and the least solution to $\rho(X_{f_i})$ equations, respectively.

The following theorem shows that the non-termination algorithm based on nt-otf is sound and computes only the necessary part of natural semantics.

Theorem 4. *Let $NT_\uparrow$ and $\rho_\downarrow$ be the greatest solution for NT and the least solution for ρ in system* nt-otf*, respectively. Then,*

1. *$NT_\uparrow$ is the greatest solution to the equation* non-term *on $K^{\mathtt{m}}$;*
2. *$\forall i \in [1..n] \cdot \rho_\downarrow(X_{f_i}) \subseteq [\![f_i]\!]$;*
3. *for any ρ, if $NT_\uparrow$ is the greatest solution to the NT equations in* nt-otf *w.r.t. ρ, then $\forall i \in [1..n] \cdot \rho_\downarrow(X_{f_i}) \subseteq \rho(X_{f_i})$.*

As in Theorem 3, part 1 of Theorem 4 shows soundness of non-termination, and parts 2 and 3 – soundness and necessity of computation of natural semantics, respectively.

Unlike reachability, non-termination requires different fixpoint solutions for NT and ρ, and thus the order of computation can influence the result. For example, consider checking $EG(pc \neq \mathbf{ex}_1)$ on $\mathtt{EX}_1$. Initially, lines 7, 8, and 9 are associated with all the valuations on x and y, i.e., $NT(7) = NT(8) = NT(9) = \Sigma_{\{x,y\}}$, and $\rho(f_2)$ is empty, which is not the partial semantics of f_2 restricted to $NT(9)$. If the computation of NT proceeds along the function call `y:=f2(y)` using the initial value of $\rho(f_2)$, $NT(8)$ is assigned $\emptyset$. Eventually, $NT(7) = NT(8) = NT(9) = \emptyset$, i.e., the algorithm incorrectly concludes that any execution starting at lines 7, 8 or 9 terminates.

The correct order for computing the solution is such that the pre-image of a set Q w.r.t. a function call to f has to be delayed until the derivation of $\rho(X_f)$ w.r.t. Q is finished. Nonetheless, since this order is only restricted to **func-call** edges, the order of the computation elsewhere can be arbitrary. This can be used to avoid deriving "full" natural semantics. Going back to the previous example, one can first compute NT along all edges except for **func-call** edges, which will assign $NT(9)$ with $x = 2 \wedge y \leq 0$, and then compute natural semantics of f_2 restricted to the post-condition $z \leq 0$. Similarly, although initially $NT(6)$ is assigned $\Sigma_{\{x,y\}}$, $NT(6) = (x = 2 \wedge y \leq 0)$ after the initial computation of NT, which means that only partial natural semantics of f_2 restricted to the post-condition $z = 2$ is needed.

In this section, we have presented mixed semantics – a stack-free operational semantics of PL, and showed how it can be used to check reachability and non-termination of programs with a finite data domain. Although the use of such semantics is not new, our formalization provides a basis for a tight integration between abstraction and analysis, which is described in the next section.

5 Abstract Reachability and Non-termination Analysis

Here, we follow the framework of abstract interpretation (AI) [12] to derive an abstract version of the concrete analysis described in Sec. 4. To do so, we require two abstract domains: abstract sets A_s whose elements approximate sets in $\mathbf{2}^{\Sigma}$, and abstract relations A_r whose elements approximate relations in $\mathbf{2}^{\Sigma\times\Sigma}$. These domains must be equipped with abstract version of all of the operations used in equations reach-otf and nt-otf. Finally, the framework of AI ensures that the solution to an abstract equation is an approximation of the solution to the corresponding concrete equation. In what follows, we identify the necessary abstract operations on A_s and A_r, and then show how to adapt predicate abstraction for our algorithm.

Abstract Domains and Operations. The domain of abstract sets A_s must be equipped with a set union $\cup$ (used in the reachability computation) and equality (to detect the fixpoint convergence). The domain of abstract relations A_r must be equipped with (a) a pre-image operator to convert abstract relations to abstract transformers over A_s, (b) asynchronous and sequential compositions of abstract relations (used in natural semantics), (c) scope extension (used to instantiate a function call using natural semantics of a function), and (d) equality (to detect the fixpoint convergence). Furthermore, we need an *assume* operator that maps an abstract set to a corresponding abstract relation; and, to apply the abstraction directly to the source code, a computable version of abstract base semantics $[\![\cdot]\!]_\alpha$ that maps each atomic statement S to an abstract relation that approximates $[\![S]\!]$ (the concrete semantics of S).

Predicate Abstraction. In the rest of this section, we show how the domain of predicate abstraction [16, 5, 18] can be extended with the necessary abstract operations to yield abstract reachability and non-termination algorithms.

Predicate abstraction provides domains for abstracting elements, sets, and relations of valuations. Let V be a set of variables, and $\mathcal{P}$ be a set of predicates over V. The *elementary* domain of predicate abstraction over $\mathcal{P}$, denoted $\Theta_{\mathcal{P}}$, is the set of all conjunctions of literals over $\mathcal{P}$. For example, if $\mathcal{P} = \{x > 0, x < y\}$, then $\neg(x > 0)$ and $(x > 0) \wedge \neg(x < y)$ are in $\Theta_{\mathcal{P}}$. An element of $\theta \in \Theta$ approximates any valuation $\sigma \in \Sigma_V$ that satisfies all literals in θ. For example, $\sigma = \langle x \mapsto 2, y \mapsto 2\rangle$ is approximated by $x > 0$, and is also approximated by $(x > 0) \wedge \neg(x < y)$ more precisely.

The elementary domain is lifted to sets and relations in an obvious way: sets over Θ represent concrete sets, and relations over Θ – concrete relations. This extension can be either *over-* or *under-approximating*, i.e., a collection of concrete valuations corresponding to an abstract set either over-approximates or under-approximates a concrete set. The over- and under-approximating interpretations can also be combined into a single *exact*-approximation using sets and relations over Belnap logic [18].

Abstract versions of set union, set and relation equality, pre-image, and base semantics for over-approximating predicate abstraction have been defined (e.g., [5]). For example, if X and Y are two *abstract* sets, their abstract union is $X \cup_\alpha Y \triangleq \lambda z \cdot X(z) \vee Y(z)$. In [18, 17], we show that these operations also naturally extend to under-approximating and exact predicate abstractions. In the latter case, conjunctions and disjunctions, e.g., $\vee$ in the definition of $\cup_\alpha$, are interpreted in Belnap logic. We define the missing abstract relational operations $assume_\alpha$, asynchronous ($\vee_\alpha$), and se-

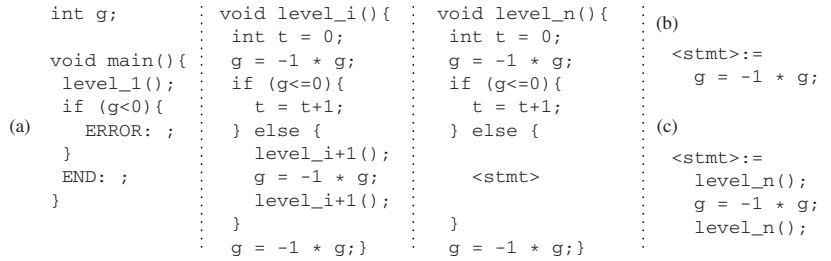

Fig. 3. (a) The template for experiments. (b) <stmt> for template **T1**(n). (c) <stmt> for **T2**(n).

quential ($\circ_\alpha$) compositions similarly, using the corresponding definitions from Sec. 2, e.g., if r_1 and r_2 are abstract relations, then their abstract asynchronous composition is $r_1 \vee_\alpha r_2 \triangleq \lambda s, t \cdot r_1(s,t) \vee r_2(s,t)$, where $\vee$ is interpreted in Boolean logic for over- and under-approximating abstraction, and in Belnap logic for exact abstraction.

In concrete semantics, scope extension is used to extend a relation to additional variables. That is, if r is a relation of type $U \to V$, then $(U \to U)r$ is an extension of r to variables in $U \setminus V$. In the abstract semantics, relations are defined over predicates; thus, abstract scope extension must extend a relation to additional predicates. To do this, we assume that the elementary abstract domain Θ corresponding to U can be decomposed into two independent abstract domains: one for V and the other – for $U \setminus V$, i.e., Θ is defined using predicates that either range only over V, or only over $U \setminus V$. Then, abstract scope extension $(\cdot \to \cdot)_\alpha$, defined as in Table 1, is a sound approximation of concrete scope extension.

Theorem 5. *Abstract operations assume$_\alpha$, $\vee_\alpha$, $\circ_\alpha$, and $(\cdot \to \cdot)_\alpha$ as defined above are sound approximations of their concrete counterparts.*

In the context of our on-the-fly algorithms, the assumption on abstract scope extension means that predicates that are used to abstract valuations at a return location l of a function call $\mathbf{x} := f(\mathbf{a})$ are either defined only over $\mathbf{x}$, or only over other variables in the scope of l. For example, predicates $x = 2$ and $y \leq 0$ can be used to abstract valuations at line 6 in the program $\texttt{EX}_1$, but predicate $x > y$ cannot. This is not a severe restriction in practice since a function can always be extended to accept additional parameters and return them without modification.

To summarize, both over- and under-approximating predicate abstractions can be used to soundly abstract reachability and non-termination analysis. The choice depends on the desired algorithm. For example, over-approximation is necessary to establish unreachability, whereas under-approximation – to establish non-termination. Since exact predicate abstraction combines them, it can be used for both verification and refutation.

6 Experiments

The technique described in this paper has been implemented in our symbolic software model checker YASM [19]. YASM is written in JAVA; it uses CVC Lite [7] to approximate program statements and CUDD [27] as a decision diagram engine. We have also

Table 3. Experimental results: overall analysis time in seconds

	T1(n)	**T2**(n)	
n	EF ($pc =$ ERROR) (reach)	EG ($pc \neq$ END) (non-terminate)	$\neg EF$ ($pc =$ ERROR) (unreach)
20	6.5	4.9	4.3
50	11.7	8.9	6.3
100	20.3	20.3	11.1
200	36.7	25.2	27.6
300	47.6	34.4	42.1
400	68.1	43.2	64.5
500	105.2	60.6	86.6

extended our proof-based refinement approach [17] to handle natural semantics of functions. In the rest of this section, we report on a preliminary evaluation of this implementation. All of the experiments have been conducted on a 2xP4Xeon-3.6GHz server and are available at `http://www.cs.toronto.edu/fm/yasm/yasm-tests.zip`. Our experiments demonstrate YASM's ability to analyze reachability and non-termination of recursive programs using exact-approximation. In summary:

1. We run YASM on template programs similar to those in the BEBOP and MOPED benchmarks. The experiment shows that the analysis time for *both* reachability and non-termination increases linearly w.r.t. the number of functions in a program.
2. We show that abstract analysis based on exact-approximation supports both verification and refutation.
3. We compare YASM with MOPED and VERA (BEBOP does not do non-termination), and show that YASM can prove non-termination of the original buggy `Quicksort` algorithm, whereas MOPED and VERA cannot.

To evaluate the reachability algorithm, we have used the template program **T1**(n) which is a variant of the one used for BEBOP in [6]. **T1**(n) is the result of replacing `<stmt>` in the template shown in Fig. 3(a) with the statements in Fig. 3(b). It consists of a `main` function and n sub-functions, where `main` calls `level_1`, and `level_i` calls `level_i+1` twice if the global variable `g` is positive. Since `g` is not initialized, its initial value is arbitrary. Although this program has no recursion, inlining function calls increases its size exponentially, making the analysis infeasible for a sufficiently large n. We checked the reachability property *EF* ($pc =$ ERROR) with values of n ranging between 20 and 500, and measured the *overall* analysis time (including parsing, abstraction, model-checking, and refinement). The results are shown in the second column of Table 3. Since our technique analyzes each function separately, the analysis time increases linearly w.r.t. the number of functions (n), as expected. In all these cases, YASM was successful in proving reachability, and discovered predicates $g < 0$, $g > 0$ and $g \leq 0$. While the template **T1**(n) is similar to the one used in [6], there is a fundamental difference: BEBOP assumes an over-approximating abstract semantics of Boolean programs and cannot *conclusively verify* that the ERROR label is reachable with these predicates. YASM uses exact-approximation which results in a conclusive analysis.

We also checked the template program **T2**(n), obtained by replacing `<stmt>` in the template in Fig. 3(a) with statements in Fig. 3(c). Non-termination and unreachability results are presented in the third and fourth columns of Table 3, respectively. As expected, the analysis time increases linearly with the number of functions.

(a)
```
void main (){
 int mx, my;
 ack (mx, my);
 END:; }

int ack (int x, int y){
 int r1,  n;
 if (x > 0) {
   if (y > 0) {
     y = y - 1;
     n = ack (x, y);
   } else { n = 1; }
   r1 = ack (x, n);
   return r1;
 } else {
   r1 = y + 1;
   return r1;
 }}
```

(b)
```
void main(){
 int x;
 foo(x);
 while(x!=0) {
   if (x<0) {
     x = -1 * x;
     x = x+2;
   } else {
     x = -1 * x;
     x = x+3;
   }}
 END: ;}

void foo (int y){
 y = -1 * y;
 if (y < 0) {
   foo (y);
 }}
```

(c)
```
void main (){
 int mleft, mright;
 quicksort (mleft, mright);
 END:;}

void quicksort (int left, int right){
 int lo, hi;
 if (left >= right) return;
 lo = left; hi = right;
 while (lo <= hi) {
   if (nondet()) {
     lo = lo+1;
   } else {
     if(lo!=left || hi!=right)
       hi = hi-1;
   }}
 quicksort (left, hi);
 quicksort (lo, right); }
```

Fig. 4. Non-terminating programs: (a) `Ack`; (b) `Shift`; (c) Buggy `Quicksort`

For non-termination, we have also applied YASM to several examples inspired by [10], in particular, on programs `Ack` and `Shift`, shown in Fig. 4(a) and (b), respectively. YASM was able to automatically prove non-termination of `Ack` in 2.1 seconds and discovered predicates $y > 0$, $n > 0$, $x > 0$, $mx > 0$ and $my > 0$. Analysis of `Shift` took 1.9 seconds and yielded predicates $y < 0$, $x < 0$, $x > 3$, $x = 0$ and $x = 3$. Finally, we have compared YASM to MOPED [14] and VERA [1] on the buggy `Quicksort` example from [14] in Fig. 4(c), where `nondet()` represents non-deterministic choice. YASM has established non-termination of `Quicksort` in 10 seconds, finding 7 predicates. Note that both MOPED and VERA only apply to programs with finite data domain, and the analysis in [1] and [14] had to restrict the number of bits used by each variable, while YASM did not need any such restriction.

7 Conclusion and Related Work

This paper presented a model-checking technique for analysis of reachability and non-termination properties of recursive programs. The technique is based on a stack-free mixed operational semantics of programs that uses natural semantics and non-determinism to eliminate the call stack while preserving stack-independent properties. We show how to compute only the necessary part of natural semantics during the analysis, leading to on-the-fly algorithms for analysis of reachability and non-termination of programs with finite data domains. We then use the framework of abstract interpretation [12] to combine our algorithms with data abstractions, making them applicable to programs with infinite data domains as well. Although we specialize our approach to predicate abstraction, we believe that it can be extended to other abstract domains as well. We have implemented a combination of this approach with exact predicate abstraction in YASM [19] which supports both verification and refutation of properties. Our experiments indicate that YASM scales to programs with a large number of functions and is able to establish non-termination of non-trivial (although small) examples. In particular, we were able to automatically prove non-termination of `Ack` [10] and `Quicksort` [14] without any restrictions on the data domain.

In the terminology of interprocedural program analysis [24], our approach is *functional* since it uses natural semantics to handle function calls. Most other model-checking approaches for recursive programs (e.g., [23, 6, 1]) are functional as well, and only compute the necessary part of natural semantics. Our reachability algorithm can be seen as a pre-image-based variant of the RHS algorithm [23], as implemented in BEBOP [6].

Both MOPED [14] and VERA [1] can check non-termination of programs with finite data domains. Their algorithms are comparable with our non-termination algorithm. However, it is unclear how to combine their techniques with an arbitrary abstraction, whereas it is quite natural in our approach. Note that an ability to detect non-termination of over-approximating Boolean programs is of limited utility since over-approximation often introduces *spurious* non-terminating computations. Thus, non-termination of an over-approximation says nothing about non-termination of the concrete program.

Jeannet and Serwe [20] apply abstract interpretation to derive abstract analysis of recursive programs by different abstractions of the call stack. Their approach is also parameterized by an arbitrary data abstraction. However, the authors restrict their attention to reachability (i.e., invariance) properties, and do not report on an implementation.

Our interest in non-termination is motivated by the work on *termination* (e.g., [10]). We view our approach as complementary to that. As illustrated by our experiments, YASM can prove non-termination of non-trivial programs. However, its ability to prove termination is limited to cases where termination can be established by a constant ranking function. In the future, we plan to investigate how the strengths of the two approaches can be combined in a single algorithm.

In this paper, we have restricted our attention to stack-independent properties. We hope to extend our approach to a more general class of properties, e.g., the ones expressible in CARET [2]. Finally, the refinement strategies that are currently implemented in YASM were originally developed for reachability analysis only. While they were sufficient for our non-termination experiments, we believe that strategies specifically tailored to the non-termination analysis are essential for scaling the tool to large programs.

References

1. Alur, R., Chaudhuri, S., Etessami, K., Madhusudan, P.: On-the-Fly Reachability and Cycle Detection for Recursive State Machines. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 61–76. Springer, Heidelberg (2005)
2. Alur, R., Etessami, K., Madhusudan, P.: A Temporal Logic of Nested Calls and Returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)
3. Ball, T.: Formalizing Counterexample-driven Refinement with Weakest Preconditions. Tech. Report 134, MSR (2004)
4. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for Falsification. In: Proceedings of CAV 2005. LNCS, vol. 3376, pp. 67–81. Springer, Heidelberg (2005)
5. Ball, T., Podelski, A., Rajamani, S.: Boolean and Cartesian Abstraction for Model Checking C Programs. STTT 5(1), 49–58 (2003)
6. Ball, T., Rajamani, S.: Bebop: A Symbolic Model Checker for Boolean Programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)

7. Barrett, C., Berezin, S.: CVC Lite: A New Implementation of the Cooperating Validity Checker. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004)
8. Bouajjani, A., Esparza, J., Maler, O.: Reachability Analysis of Pushdown Automata: Application to Model-Checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
9. Bruns, G., Godefroid, P.: Model Checking Partial State Spaces with 3-Valued Temporal Logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999)
10. Cook, B., Podelski, A., Rybalchenko, A.: Termination Proofs for System Code. In: Proceedings of PLDI 2006, pp. 415–426 (2006)
11. Cousot, P.: Asynchronous Iterative Methods for Solving a Fixed Point System of Monotone Equations in a Complete Lattice. Research report, Univ. of Grenoble (September 1977)
12. Cousot, P., Cousot, R.: Abstract Interpretation Frameworks. J. of Logic and Computation 2(4), 511–547 (1992)
13. Dams, D., Gerth, R., Grumberg, O.: Abstract Interpretation of Reactive Systems. ACM Transactions on Programming Languages and Systems 2(19), 253–291 (1997)
14. Esparza, J., Schwoon, S.: A BDD-Based Model Checker for Recursive Programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
15. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based Model Checking using Modal Transition Systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154. Springer, Heidelberg (2001)
16. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
17. Gurfinkel, A., Chechik, M.: Why Waste a Perfectly Good Abstraction? In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 212–226. Springer, Heidelberg (2006)
18. Gurfinkel, A., Wei, O., Chechik, M.: Systematic Construction of Abstractions for Model-Checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 381–397. Springer, Heidelberg (2005)
19. Gurfinkel, A., Wei, O., Chechik, M.: YASM: A Software Model-Checker for Verification and Refutation. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 170–174. Springer, Heidelberg (2006)
20. Jeannet, B., Serwe, W.: Abstracting Call-Stacks for Interprocedural Verification of Imperative Programs. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116. Springer, Heidelberg (2004)
21. Larsen, K., Thomsen, B.: A Modal Process Logic. In: Proceedings of LICS 1988, pp. 203–210. IEEE Computer Society Press, Los Alamitos (1988)
22. Nielson, H., Nielson, F.: Semantics with Applications: A Formal Introduction. Wiley Professional Computing, Chichester (1992)
23. Reps, T.W., Horwitz, S., Sagiv, M.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: Proceedings of POPL 1995, pp. 49–61 (1995)
24. Sharir, M., Pnueli, A.: Program Flow Analysis: Theory and Applications. In: Two Approaches to Interprocedural Data Flow Analysis, pp. 189–233. Prentice-Hall, Englewood Cliffs (1981)
25. Shoham, S., Grumberg, O.: A Game-Based Framework for CTL Counter-Examples and 3-Valued Abstraction-Refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 275–287. Springer, Heidelberg (2003)
26. Shoham, S., Grumberg, O.: Monotonic Abstraction-Refinement for CTL. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 546–560. Springer, Heidelberg (2004)
27. Somenzi, F.: CUDD: CU Decision Diagram Package Release (2001)