Finding Recurrent Sets with Backward Analysis and Trace Partitioning

Alexey Bakhirkin and Nir Piterman

University of Leicester, Department of Computer Science

Abstract. We propose an abstract-interpretation-based analysis for recurrent sets. A recurrent set is a set of states from which the execution of a program cannot or might not (as in our case) escape. A recurrent set is a part of a program's non-termination proof (that needs to be complemented by reachability analysis). We find recurrent sets by performing a potentially over-approximate backward analysis that produces an initial candidate. We then perform over-approximate forward analysis on the candidate to check and refine it and ensure soundness. In practice, the analysis relies on trace partitioning that predicts future paths through the program that non-terminating executions will take. Using our technique, we were able to find recurrent sets in many benchmarks found in the literature including some that, to our knowledge, cannot be handled by existing tools. In addition, we note that typically, analyses that search for recurrent sets are applied to linear under-approximations of programs or employ some form of non-approximate numeric reasoning. In contrast, our analysis uses standard abstract-interpretation techniques and is potentially applicable to a larger class of abstract domains (and therefore – programs).

1 Introduction

Termination is a fundamental property of software routines. The majority of code is required to terminate (e.g., dispatch routines of device drivers or other event-driven code, GPU programs) and the existence of a non-terminating behavior is a severe bug that might freeze a device, an entire system [1], or cause a multi-region cloud service disruption [2]. The problem of proving *termination* has seen much attention lately [15, 16, 33], but the techniques are sound and hence necessarily incomplete. That is, failure to prove termination does not imply the existence of non-terminating behaviors. Hence, proving *non-termination* is an interesting complementary problem.

Several modern analyses [8, 13, 14, 26] characterize non-terminating behaviors with a notion of *recurrent set*, i.e., a set of states from which an execution of the program or fragment cannot or might not escape (there exist multiple definitions). In this paper, we focus on the notion of an *existential recurrent set* – a set of states, s.t., from every state in the set there exists at least one non-terminating execution. Typically, the analyses that find existential recurrent sets and/or prove non-termination are applied to linear *under*-approximations of programs [13] and/or employ some form of non-approximate numeric reasoning, e.g., using an SMT-solver as in [12], or applying Farkas' lemma as (to our knowledge) in [10]. This allows the analyses to produce genuine recurrent sets. In the context of abstract interpretation (that may go beyond numeric reasoning), under-approximation is problematic. For example, as we show later, fixed-point characterization of an existential recurrent set involves set union, and in most abstract domains it is hard to define an under-approximate join operation.

In this paper, we propose a sound abstract-interpretation-based analysis that finds existential recurrent sets via approximate reasoning. The proposed analysis works in two steps. First, we perform approximate (potentially, over-approximate) backward analysis to find a *candidate recurrent set*. An important technique that allows finding successful candidates is *trace partitioning* (for trace partitioning in forward analysis, see [29]). Then, we perform over-approximate forward analysis on the candidate to check and refine it and ensure soundness. We define the analysis for imperative programs without procedures, and we apply it separately for every loop of the program (i.e., every strongly connected component of the program graph). We evaluated the analysis on the test set [3] of Invel [35], on non-terminating programs from the SV-COMP 2015 [4] termination category, and on a set of non-deterministic numeric programs that we produced ourselves. In this paper, we make a number of assumptions on the memory domain. In particular, we assume that there exists a meet operation that allows backward analysis to build a descending chain; then, we use lower widening to ensure convergence of backward analysis. Non-numeric domains may employ different techniques. For example, in shape analysis with 3-valued logic [31], convergence is due to the use of a finite domain of *bounded structures*. Our backward analysis would need to be modified to be applicable to this and similar domains.

Finally, we note that finding a recurrent set is a *sub-problem* of proving non-termination (in this paper, by proving non-termination we mean proving the existence of at least one non-terminating execution). To prove non-termination, we would need to show that a recurrent set is reachable from the program entry which we *do not* address in this paper for practical reasons. In theory, our analysis may find a recurrent set in any program or fragment (not necessarily strongly connected), and if the inferred set contains an initial program state, this proves the existence of non-terminating behaviours. In practice, we have so far obtained satisfactory results only with finding recurrent sets of individual loops. There also exists work on showing feasibility of abstract counterexamples (including, for non-numeric abstract domains [9]), and techniques from that area would also be applicable to show reachability of a recurrent set.

2 Background

We use 1 and 0 to mean logical truth and falsity respectively. We use Kleene's 3-valued logic [25] to represent truth values of state formulas in abstract states and sets of concrete states. The logic uses a set of three values $\mathcal{K} = \{0, {}^1\!/_2, 1\}$ meaning *false*, *maybe*, and *true* respectively. $\mathcal{K}$ is arranged in partial *information order* $\sqsubseteq_{\mathcal{K}}$, s.t. 0 and 1 are incomparable, $0 \sqsubseteq_{\mathcal{K}} {}^1\!/_2$, and $1 \sqsubseteq_{\mathcal{K}} {}^1\!/_2$. For $k_1, k_2 \in \mathcal{K}$, the least upper bound $\sqcup_{\mathcal{K}}$ is s.t. $k_1 \sqcup_{\mathcal{K}} k_2 = k_1$ if $k_1 = k_2$, and ${}^1\!/_2$ otherwise. For a lattice $\mathcal{L}$ ordered by $\preccurlyeq$ and a monotonic function $F : \mathcal{L} \to \mathcal{L}$, we use $\mathrm{lfp}_{\preccurlyeq} F$ to denote the least fixed point of F and $\mathrm{gfp}_{\preccurlyeq} F$ to denote the greatest fixed point.

States, Statements, and Programs. Let $\mathbb{M}$ be the set of *memory states*. A memory state may map program variables to their values, describe the shape of the heap, etc.

A *memory-state formula* θ denotes a set of memory states $[\![\theta]\!] \subseteq \mathbb{M}$. In this paper, the formulas will usually be conjunctions of linear inequalities over the program variables. E.g., the formula $x > 0$ will denote the set of memory states where x is positive. We say that a memory state $m \in \mathbb{M}$ satisfies θ if $m \in [\![\theta]\!]$. For a memory-state formula θ and a set of memory states $M \subseteq \mathbb{M}$, the *value* of θ over M is defined as: $\mathrm{eval}(\theta, M) = 1$ if $M \subseteq [\![\theta]\!]$; $\mathrm{eval}(\theta, M) = 0$ if $M \neq \varnothing \wedge M \cap [\![\theta]\!] = \varnothing$; $\mathrm{eval}(\theta, M) = {}^1\!/_2$ otherwise. That is, a formula evaluates to 1 in a *set* of memory states if all the memory states in the set satisfy the formula; to 0 if the set is non-empty and no memory states in the set satisfy the formula; and to ${}^1\!/_2$ if some memory states satisfy the formula and some do not.

Let $\mathbb{C}$ be the set of *atomic statements*. For a statement $C \in \mathbb{C}$, its *input-output relation* is $T_{\mathbb{M}}(C) \subseteq \mathbb{M} \times \mathbb{M}$. A pair of memory states $(m, m') \in T_{\mathbb{M}}(C)$, iff it is possible to produce m' by executing C in m. We assume that $\mathbb{C}$ includes (but is not limited to):

(i) a passive statement *skip* with $T_{\mathbb{M}}(\mathit{skip}) = \{(m, m) \mid m \in \mathbb{M}\}$; and

(ii) an assumption statement $[\theta]$ for every memory-state formula θ, with $T_{\mathbb{M}}([\theta]) = \{(m, m) \mid m \in [\![\theta]\!]\}$. The main use of assumption statements is to represent branch and loop conditions.

What other the statements are in $\mathbb{C}$ depends on the class of programs we're working with; e.g., for numeric programs, $\mathbb{C}$ may include assignments of the form $x = \mathit{expr}$.

We assume that for other atomic statements, their input-output relations are given. We require that for every non-assumption statement $C \in \mathbb{C}$, the input-output relation of C is left-total, i.e., for every memory state $m \in \mathbb{M}$, there exists a successor state $m' \in \mathbb{M}$, s.t., $(m, m') \in T_{\mathbb{M}}(C)$. In this paper, we do not discuss the analysis of unsafe programs, but if executing C in some $m \in \mathbb{M}$ may *fail*, we assume that there exists a distinguished *error memory state* ε, s.t. $(m, \varepsilon) \in T_{\mathbb{M}}(C)$. In this paper, we work with programs that manipulate numeric variables, most often (but not necessarily) integer-valued. Thus, given the set of program variables $\mathbb{V}$, we can assume $\mathbb{M} = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$.

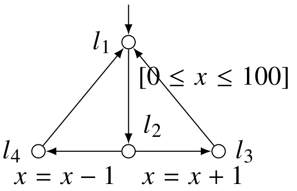

Fig. 1. Loop with non-deterministic branching.

A *program* $\mathbb{P}$ is a graph $(\mathbb{L}, \mathfrak{l}_{\vdash}, \mathbb{E}, \mathfrak{c})$ where $\mathbb{L}$ is a finite set of *program locations* that are vertices of the graph; $\mathfrak{l}_{\vdash} \in \mathbb{L}$ is a distinguished *initial location*; $\mathbb{E} \subseteq \mathbb{L} \times \mathbb{L}$ is a set of *edges*; and $\mathfrak{c} : \mathbb{E} \to \mathbb{C}$ labels edges with atomic statements. A location without outgoing edges is a *final location*. Intuitively, an execution of the program terminates iff it reaches a final location. For a location $l \in \mathbb{L}$, the *successors* of l is the set $\mathrm{succ}(l) = \{l' \in \mathbb{L} \mid (l, l') \in \mathbb{E}\}$. Note that for $l, l' \in \mathbb{L}$, we allow at most one edge from l to l'. This simplifies the presentation, but does not restrict the allowed class of programs.

An example of a program is shown in Fig. 1. It is a loop where in every iteration, the execution makes a non-deterministic choice: whether to increment or decrement the variable x (thus, $\mathbb{V} = \{x\}$). The set of location $\mathbb{L} = \{l_1, \cdots, l_4\}$, the initial location $\mathfrak{l}_{\vdash} = l_1$. The program does not have a final location and can be assumed to be a fragment of a larger program (as discussed later, our analysis works with such fragments). Also note how we cannot have multiple edges from l_2 to l_1, and we use locations l_3 and l_4 to work around that (for the edges displayed without a label, we assume the label *skip*).

Set $\mathbb{S} = \mathbb{L} \times \mathbb{M}$ is the set of *program states*. We say that a program state $s \in \mathbb{S}$ is final iff $s = (l, m)$ for a final location $l \in \mathbb{L}$ and some memory state $m \in \mathbb{M}$. For a program $\mathbb{P}$, the *transition relation on program states* $T_{\mathbb{S}}(\mathbb{P}) \subseteq \mathbb{S} \times \mathbb{S}$ is s.t. a pair of program states $((l, m), (l', m')) \in T_{\mathbb{S}}(\mathbb{P})$ iff one of the following holds: (i) $(l, l') \in \mathbb{E}$ and $(m, m') \in T_{\mathbb{M}}(\mathfrak{c}(l, l'))$; or (ii) l is final, $l' = l$, and $m' = m$. That is, the transition relation consists of pairs of program states (s, s'), s.t. it is possible to reach s' by executing an atomic statement from s or by staying in the same final state.

Traces and Executions. To use trace partitioning, we need to be able to reason not only about memory states and locations, but also about traces. A *path* is a pair $(p, i) \in \mathbb{L}^{\mathbb{N}} \times \mathbb{N}$, where $p = \langle l_0, l_1, l_2, \ldots \rangle \in \mathbb{L}^{\mathbb{N}}$ is an *infinite* sequence of locations, and $i \geq 0$ is a (current) *position*. Intuitively, a path is a sequence of locations that is visited by a potential run of the program, together with a point in the run where we currently are. We denote the set of paths by Π. For a path $\pi = (p, i) \in \Pi$, $p_{(0)}$ and $\pi_{(0)}$ denote the first location in the path; $p_{(j)}$ and $\pi_{(j)}$ denote the $j+1$-th location.

A *trace* is a pair $(t, i) \in \mathbb{S}^{\mathbb{N}} \times \mathbb{N}$, where $t = \langle s_0, s_1, s_2, \ldots \rangle \in \mathbb{S}^{\mathbb{N}}$ is an infinite sequence of program states, and $i \geq 0$ is a (current) position. Intuitively, a trace is a sequence of program states that is visited by a potential run of the program, together with a point in the run where we currently are. We denote the set of traces by Σ. For a trace $\tau \in \Sigma$, $t_{(0)}$ and $\tau_{(0)}$ denote the first state of the trace; $t_{(j)}$ and $\tau_{(j)}$ denote the $j+1$-th state. For a location $l \in \mathbb{L}$, the set of all traces *at* l is $\Sigma|_l = \{(t, i) \in \Sigma \mid \exists m \in \mathbb{M}.\ t_{(i)} = (l, m)\}$. The set of all traces at l and position i is $\Sigma|_{l,i} = \{(t, i) \in \Sigma \mid \exists m \in \mathbb{M}.\ t_{(i)} = (l, m)\}$. For example, $\Sigma|_{\mathfrak{l}_{\vdash},0}$ is the set of traces, s.t. they start at the initial program location $\mathfrak{l}_{\vdash}$, and the current position is 0. For a trace $\tau \in \Sigma$, its path $\mathfrak{p}(\tau) \in \Pi$ is produced by removing information about the memory states. For $\tau = (\langle (l_0, m_0), (l_1, m_1), (l_2, m_2), \ldots \rangle, i) \in \Sigma$, $\mathfrak{p}(\tau) = (\langle l_0, l_1, l_2, \ldots \rangle, i) \in \Pi$. We say that a trace is *terminating* iff there exists $j \geq 0$, a final location $l \in \mathbb{L}$, and a memory state $m \in \mathbb{M}$, s.t., for every $k \geq j$, $\tau_{(k)} = (l, m)$. We say that a trace is non-terminating iff it is not terminating.

Given a program $\mathbb{P}$, not every trace can be produced by it. A trace $\tau \in \Sigma$ is a *semi-execution* of $\mathbb{P}$ iff for every $j \geq 0$, $(\tau_{(j)}, \tau_{(j+1)}) \in T_{\mathbb{S}}(\mathbb{P})$. A trace $\tau \in \Sigma$ is an *execution*, if it is a semi-execution and $\tau_{(0)} = (\mathfrak{l}_{\vdash}, m)$ for some memory state $m \in \mathbb{M}$. Intuitively, an execution, as its first component, has a sequence of program states that is produced by starting in the initial program location in some memory state, and running the program either infinitely (producing a non-terminating execution) or until it terminates in a final location (producing a terminating one). For the program in Fig. 1, we can produce a non-terminating execution by, e.g., alternating the increment and decrement of x: $(\langle ((l_1, x \mapsto 0), (l_2, x \mapsto 0), (l_3, x \mapsto 1), (l_1, x \mapsto 1), (l_2, x \mapsto 1), (l_4, x \mapsto 0))^{\mathbb{N}} \rangle, i)$. A trace $(t, i) \in \Sigma$ is an *execution prefix* iff $t_{(0)} = (\mathfrak{l}_{\vdash}, m)$ for some memory state $m \in \mathbb{M}$, and for every j, s.t. $0 \leq j < i$, $(t_{(j)}, t_{(j+1)}) \in T_{\mathbb{S}}(\mathbb{P})$. Intuitively, for an *execution prefix* (t, i), the prefix of t up to position i is produced by starting in the initial location in some memory state and making i steps through the program. A trace $(t, i) \in \Sigma$ is an execution postfix iff for every $j \geq i$, $(t_{(j)}, t_{(j+1)}) \in T_{\mathbb{S}}(\mathbb{P})$. We lift the program transition relation to traces and paths. The transition relation on traces is $T_{\Sigma}(\mathbb{P}) = \{((t, i), (t, i{+}1)) \in \Sigma \times \Sigma \mid (t_{(i)}, t_{(i+1)}) \in T_{\mathbb{S}}(\mathbb{P})\}$. The transition relation on paths is $T_{\Pi}(\mathbb{P}) = \{((p, i), (p, i{+}1)) \in \Pi \times \Pi \mid (p_{(i)}, p_{(i+1)}) \in \mathbb{E}\}$.

Non-termination analysis and set-of-states abstraction. For a set of traces $S \subseteq \Sigma$, the *closed subset* $(\!|S|\!) = \{(t,i) \in S \mid \forall j \geq 0.\ (t,j) \in S\}$. That is, $(\!|S|\!)$ is the largest subset of S closed under shifting the position.

Given some set S_0 and a transition relation $T \subseteq S_0 \times S_0$, the *post-condition* and *pre-condition* of a set $S \subseteq S_0$ via T are the sets:

$$\mathrm{post}(T,S) = \{s' \in S_0 \mid \exists s \in S.\ (s,s') \in T\}, \quad \mathrm{pre}(T,S) = \{s \in S_0 \mid \exists s' \in S.\ (s,s') \in T\}$$

For a program $\mathbb{P}$, *non-termination* analysis of $\mathbb{P}$ is the greatest fixed point:

$$\mathrm{gfp}_{\subseteq}\ \lambda X.((\bigcup\{\Sigma|_l \text{ for non-final } l \in \mathbb{L}\}) \cap \mathrm{pre}(T_\Sigma(\mathbb{P}), X)) \tag{1}$$

Lemma 1. *For a program $\mathbb{P}$ the closed subset of its non-termination analysis gives the set of all non-terminating semi-executions of the program.*

Proof idea. Intuitively, non-termination analysis retains non-terminating execution postfixes. Taking closed subset keeps only the traces that also are execution prefixes: if (t,i) is in the closed subset, then for every j, s.t., $0 \leq j < i$, (t,j) must be in the closed subset and thus must be an execution postfix, i.e., (t,i) must be a semi-execution. □

Note that usually, a pre-condition through the whole program is computed as a *union* of pre-conditions through the program statements. This makes it hard to define a sound computable non-termination analysis, since in most abstract domains it is hard to define an under-approximate join operation.

For a set of traces $S \subseteq \Sigma$, the *set-of-states abstraction* $\alpha_\mathrm{s}(S) \in \mathbb{S}$ collects current program states of every trace: $\alpha_\mathrm{s}(S) = \{s' \in \mathbb{S} \mid \exists (t,i) \in S.\, t_{(i)} = s'\}$. The corresponding concretization γ_s, for $S' \subseteq \mathbb{S}$ produces the set of traces that have an element of S' at the current position: $\gamma_\mathrm{s}(S') = \{(t,i) \in \Sigma \mid t_{(i)} \in S'\}$. For $S' \subseteq \mathbb{S}$, $(\!|\gamma_\mathrm{s}(S')|\!) = \{(t,i) \in \Sigma \mid \forall j \geq 0.\ t_{(j)} \in S'\}$. This is the set of traces that only visit program states from S'.

Existential recurrent set. For a program $\mathbb{P}$, a set of program states $S_\exists \subseteq \mathbb{S}$ is an *existential recurrent set* if for every $s \in S_\exists$, s is not final and there exists $s' \in S_\exists$, s.t., $(s,s') \in T_\mathbb{S}(\mathbb{P})$. Intuitively, this is a set of program states, from which the program *may* run forever. Note that by this definition, an empty set is trivially existentially recurrent. The authors of [13] use a similar (but stronger) notion of *open recurrent set*, requiring that all the states in the open recurrent set are reachable. In this paper, by just *recurrent set* we mean existential recurrent set.

Lemma 2. *Set-of-states abstraction of non-termination analysis gives the largest existential recurrent set.*

Proof idea. Intuitively a recurrent set $S_\exists$ is s.t. from every element of $S_\exists$ we can start a non-terminating semi-execution that only visits elements of $S_\exists$. Non-termination analysis produces the set of all non-terminating execution postfixes, and by applying set-of-states abstraction to it, we produce the set of all program states from which we can start a non-terminating semi-execution, i.e., the maximal recurrent set. □

The problem of finding a recurrent set is a sub-problem of proving non-termination. To prove non-termination (i.e., the existence of at least one non-terminating execution), we would need to find a recurrent set and show that it is reachable from an initial state. In this paper, though, we focus on finding a recurrent set only.

Memory and path abstraction. From the set-of-states abstraction of an analysis, one can produce a computable over-approximate analysis by performing further memory abstraction, which is standard in abstract interpretation. We introduce *memory abstract domain* $\mathbb{D}_{\mathrm{m}}$, with least element $\bot_{\mathrm{m}}$, greatest element $\top_{\mathrm{m}}$, partial order $\sqsubseteq_{\mathrm{m}}$, and join $\sqcup_{\mathrm{m}}$. Every *element*, or abstract memory state, $a \in \mathbb{D}_{\mathrm{m}}$ represents a set of memory states $\gamma_{\mathrm{m}}(a) \subseteq \mathbb{M}$. For the analysis of numeric programs, $\mathbb{D}_{\mathrm{m}}$ can be a polyhedral domain where an element is a conjunction of linear inequalities over the program variables.

We lift *concretization* to sets of abstract memory states: for $A \subseteq \mathbb{D}_{\mathrm{m}}$, $\gamma_{\mathrm{m}}(A) = \bigcup\{\gamma_{\mathrm{m}}(a) \mid a \in A\}$. We introduce over-approximate versions of post, pre, and eval, s.t. for a statement $C \in \mathbb{C}$, an element $a \in \mathbb{D}_{\mathrm{m}}$, and a memory-state formula θ,

$$\begin{aligned} &\gamma_{\mathrm{m}}(\mathrm{post}^{\mathrm{m}}(C, a)) \supseteq \mathrm{post}(T_{\mathbb{M}}(C), \gamma_{\mathrm{m}}(a)) \qquad && \mathrm{eval}^{\mathrm{m}}(\theta, a) \sqsupseteq_{\mathcal{K}} \mathrm{eval}(\theta, \gamma(a)) \\ &\gamma_{\mathrm{m}}(\mathrm{pre}^{\mathrm{m}}(C, a)) \supseteq \mathrm{pre}(T_{\mathbb{M}}(C), \gamma_{\mathrm{m}}(a)) && \end{aligned}$$

Normally, $\mathrm{eval}^{\mathrm{m}}$ is given for atomic formulas; for arbitrary formulas it is defined by induction over the formula structure, using 3-valued logical operators, possibly over-approximate w.r.t. $\sqsubseteq_{\mathcal{K}}$. In this paper, we make additional assumptions on $\mathbb{D}_{\mathrm{m}}$. We assume there exists meet operation, s.t., for $a_1, a_2 \in \mathbb{D}_{\mathrm{m}}$, $a_1 \sqcap_{\mathrm{m}} a_2 \sqsubseteq_{\mathrm{m}} a_1$ and $a_1 \sqcap_{\mathrm{m}} a_2 \sqsubseteq_{\mathrm{m}} a_2$. This allows producing descending chains in $\mathbb{D}_{\mathrm{m}}$ and performing approximation of greatest fixed points even with non-monotonic abstract transformers. If $\mathbb{D}_{\mathrm{m}}$ admits infinite descending chains, we assume there exists *lower widening* operation $\underline{\triangledown}_{\mathrm{m}}$. Similarly, if $\mathbb{D}_{\mathrm{m}}$ admits infinite ascending chains, we assume there exists *widening* operation $\triangledown_{\mathrm{m}}$. To produce a standard over-approximate analysis one transitions to the domain $\mathbb{L} \to \mathbb{D}_{\mathrm{m}}$, where every element represents a set of program states partitioned with locations.

We are going to use trace partitioning and we take an additional step to introduce what we call a *path abstract domain* $\mathbb{D}_{\mathrm{p}}$, with least element $\bot_{\mathrm{p}}$, greatest element $\top_{\mathrm{p}}$, partial order $\sqsubseteq_{\mathrm{p}}$, join $\sqcup_{\mathrm{p}}$ and meet $\sqcap_{\mathrm{p}}$. Every *element*, or abstract path, $q \in \mathbb{D}_{\mathrm{p}}$ represents a set of paths $\gamma_{\mathrm{p}}(q) \subseteq \Pi$. We introduce over-approximate versions of post and pre, s.t. for an edge $e \in \mathbb{E}$ and an element $q \in \mathbb{D}_{\mathrm{p}}$,

$$\gamma_{\mathrm{p}}(\mathrm{post}^{\mathrm{p}}(e, q)) \supseteq \mathrm{post}(T_{\Pi}(\mathbb{P})|_e, \gamma_{\mathrm{p}}(q)) \qquad \gamma_{\mathrm{p}}(\mathrm{pre}^{\mathrm{p}}(e, q)) \supseteq \mathrm{pre}(T_{\Pi}(\mathbb{P})|_e, \gamma_{\mathrm{p}}(q))$$

where $T_{\Pi}(\mathbb{P})|_e = \{((p, i), (p, i+1)) \in \Pi \times \Pi \mid (p_{(i)}, p_{(i+1)}) = e\}$, i.e., it restricts the transition relation on paths to an edge $e \in E$. For our purposes, we also assume that $\mathbb{D}_{\mathrm{p}}$ is finite, and there exists abstraction function α_{p} that, together with γ_{p} forms a Galois connection between $\mathbb{D}_{\mathrm{p}}$ and $\mathcal{P}(\Pi)$. This allows to partition memory states with elements of $\mathbb{L} \times \mathbb{D}_{\mathrm{p}}$, similarly to how a standard analysis partitions memory states with locations.

Abstract domain of the analysis. Given a memory abstract domain $\mathbb{D}_{\mathrm{m}}$ and path abstract domain $\mathbb{D}_{\mathrm{p}}$ with required properties, we can construct the abstract domain $\mathbb{D}_{\sharp} \subseteq \mathbb{D}_{\mathrm{p}} \rightharpoonup \mathbb{D}_{\mathrm{m}}$ (where $\rightharpoonup$ denotes a partial function). We require that every element $D \in \mathbb{D}_{\sharp}$ is what we call *reduced*: for every $q \in \mathrm{dom}(D)$, $q \neq \bot_{\mathrm{p}}$ and $D(q) \neq \bot_{\mathrm{m}}$; and for every pair of abstract paths $q_1, q_2 \in \mathrm{dom}(D)$, $q_1 \sqcap_{\mathrm{p}} q_2 = \bot_{\mathrm{p}}$. Intuitively, D is a collection of abstract memory states partitioned with *disjoint* abstract paths.

Idea of the construction. $\mathbb{D}_{\sharp}$ is ordered by $\sqsubseteq_{\sharp}$ point-wise, $\top_{\sharp} = \{\top_{\mathrm{p}} \mapsto \top_{\mathrm{m}}\}$, and $\bot_{\sharp}$ is the empty partial function. For every partial function $D' : \mathbb{D}_{\mathrm{p}} \rightharpoonup \mathbb{D}_{\mathrm{m}}$, we can produce a reduced element $D \in \mathbb{D}_{\sharp}$: we remove "bottoms" and then repeatedly join the pairs from D' (thinking of a function as of a set of pairs) that have non-disjoint abstract paths. From this point, it is straightforward to construct join $\sqcup_\sharp$, abstract post-condition $\mathrm{post}^\sharp(e, d)$, and abstract pre-condition $\mathrm{pre}^\sharp(e, d)$, $e \in \mathbb{E}$ and $d \in \mathbb{D}_\sharp$. When taking meet of $D_1, D_2 \in \mathbb{D}_\sharp$, we meet the tuples from D_1 and D_2 pair-wise. As both D_1 and D_2 are reduced, it follows that $D_1 \sqcap_\sharp D_2$ is reduced; and $D_1 \sqcap_\sharp D_2 \sqsubseteq_\sharp D_1$ and $D_1 \sqcap_\sharp D_2 \sqsubseteq_\sharp D_2$. Widening and lower widening are defined point-wise.

Then, we transition from $\mathbb{D}_\sharp$ to $\mathbb{D}_{\mathfrak{l}\sharp} = \mathbb{L} \to \mathbb{D}_\sharp$ in which backward analysis is performed. Such transition is standard in abstract interpretation (usually, it is from $\mathbb{D}_\mathfrak{m}$ to $\mathbb{L} \to \mathbb{D}_\mathfrak{m}$) and we do not describe it. We only note that in $\mathbb{D}_{\mathfrak{l}\sharp}$, the post-condition $\mathrm{post}^{\mathfrak{l}\sharp}(\mathbb{P}, \cdot\,)$ and pre-condition $\mathrm{pre}^{\mathfrak{l}\sharp}(\mathbb{P}, \cdot\,)$ are taken with respect to the whole program. We prefer to think of an element $D_l \in \mathbb{D}_{\mathfrak{l}\sharp}$ as of a collection of abstract program states partitioned by location and abstract path.

3 Finding a Recurrent Set

In this section we describe the main analysis steps: a backward analysis for a candidate recurrent set that is performed below the set of reachable states; followed by a forward refinement step that produces a genuine recurrent set.

We start by performing a standard forward pre-analysis of the whole program $\mathbb{P}$ to find the over-approximation of the set of reachable program states $F \in \mathbb{D}_{\mathfrak{l}\sharp}$. F is the stable limit of the sequence of $\{f_i\}_{i \geq 0}$ where $f_0 = \{\mathfrak{l}_\vdash \mapsto \top_\sharp; l \neq \mathfrak{l}_\vdash \mapsto \bot_\sharp\}$; for $i \geq 1$, $f_i = f_{i-1} \triangledown_{\mathfrak{l}\sharp} (f_{i-1} \sqcup_{\mathfrak{l}\sharp} \mathrm{post}^{\mathfrak{l}\sharp}(\mathbb{P}, f_{i-1}))$; and $\triangledown_{\mathfrak{l}\sharp}$ is a widening operator.

3.1 Backward Analysis For a Candidate

Next, we perform approximate (possibly, over-approximate) backward analysis to find candidate recurrent sets. We do it separately for every strongly connected sub-program $\mathbb{P}_s$ that represents a loop of the original program $\mathbb{P}$. More formally, we perform the analysis for every *strongly connected component* [32] $\mathbb{P}_s = (\mathbb{L}_s, \mathfrak{l}_{s\vdash}, \mathbb{E}_s, \mathfrak{c}|_{\mathbb{E}_s})$ where $\mathbb{L}_s \subseteq \mathbb{L}$; $\mathbb{E}_s \subseteq (\mathbb{L}_s \times \mathbb{L}_s) \cap \mathbb{E}$; $|\mathbb{L}_s| > 1$ or $(\mathfrak{l}_{s\vdash}, \mathfrak{l}_{s\vdash}) \in \mathbb{E}_s$ (i.e., the component represents a loop in the program); $\mathfrak{c}|_{\mathbb{E}_s}$ is the restriction of $\mathfrak{c}$ to the edges of $\mathbb{P}_s$; and $\mathfrak{l}_{s\vdash} \in \mathbb{L}_s$ is the *head* of the strongly connected component which is usually selected as the first location of the component encountered in $\mathbb{P}$ by a depth-first search. We can restrict the notion of successors to a sub-program: for $l \in \mathbb{L}_s$, $\mathrm{succ}(l)|_{\mathbb{P}_s} = \{l' \in \mathbb{L}_s \mid (l, l') \in \mathbb{E}_s\}$. Note that since $\mathbb{P}_s$ is strongly connected, it does not have final locations.

For every strongly connected sub-program $\mathbb{P}_s$, we find the candidate recurrent set $W_s \in \mathbb{D}_{\mathfrak{l}\sharp}$ as the stable limit of the sequence of elements $\{w_i\}_{i \geq 0}$ that approximates non-termination analysis below F. Here, $w_0 = F|_{\mathbb{L}_s}$ (the restriction of F to the locations of $\mathbb{P}_s$); for $i \geq 1$, $w_i = w_{i-1} \underline{\triangledown}_{\mathfrak{l}\sharp} (w_{i-1} \sqcap_{\mathfrak{l}\sharp} \mathrm{pre}^{\mathfrak{l}\sharp}(\mathbb{P}_s, w_{i-1}))$; and $\underline{\triangledown}_{\mathfrak{l}\sharp}$ is a lower widening operator. Note that we use over-approximate operations (join, backward transformers) to compute W_s, and hence W_s may over-approximate non-termination analysis and might not represent a genuine recurrent set. Although formally an element of $\mathbb{D}_{\mathfrak{l}\sharp}$ concretizes to a set of traces, we can think that W_s represents a candidate recurrent set $\alpha_\mathfrak{s}(\gamma_{\mathfrak{l}\sharp}(W_s)) = \{(l, m) \in \mathbb{S} \mid \exists q \in \mathbb{D}_\mathfrak{p}.\ m \in \gamma_\mathfrak{m}(W_s(l)(q))\}$. In the next step, we will produce a refined element $R_s \sqsubseteq_{\mathfrak{l}\sharp} W_s$ representing a genuine recurrent set.

In theory, a recurrent set does not have to be below F, but in practice, a combination of backward and forward analyses is known to be more precise than just, e.g., backward analysis [17], and we found that performing backward analysis below F (rather than below $\top_{\mathfrak{i}\sharp}$) better directs the search for a recurrent set. Intuitively, some information (e.g., conditions of assumption statements) is better propagated by forward analysis, and this information may be important to find a genuine recurrent set. Another feature important for precision is trace partitioning. We observe that for many imperative programs, non-terminating executions take a specific path through the loop. When we perform backward analysis with trace partitioning, abstract memory states in W_s are partitioned by the path through the loop that the program run would take from them. If the path domain is precise enough, s.t., (states, from which exist) non-terminating semi-executions get collected in separate partitions, the analysis is more likely to find a genuine recurrent set.

3.2 Checking and Refining the Candidate

Approximate backward analysis for every strongly connected component $\mathbb{P}_s$ of the original program, produces an element $W_s \in \mathbb{D}_{\mathfrak{i}\sharp}$, which represents a candidate recurrent set. We use over-approximate operations (join, backward transformers) to compute W_s, and hence W_s may over-approximate non-termination analysis and might not represent a genuine recurrent set. We refine W_s to a (possibly, bottom) element $R_s \sqsubseteq_{\mathfrak{i}\sharp} W_s$ representing a genuine recurrent set of $\mathbb{P}_s$ and hence of the original program $\mathbb{P}$. That is, we produce such R_s that $\forall s \in \alpha_{\mathfrak{s}}(\gamma_{\mathfrak{i}\sharp}(R_s)).\ \exists s' \in \alpha_{\mathfrak{s}}(\gamma_{\mathfrak{i}\sharp}(R_s)).\ (s, s') \in T_{\mathbb{S}}(\mathbb{P}_s)$. To do so, we define a predicate CONT, s.t. for an abstract memory state $a \in \mathbb{D}_{\mathfrak{m}}$, a set of abstract memory states $A \subseteq \mathbb{D}_{\mathfrak{m}}$, and an atomic statement $C \in \mathbb{C}$, if $\mathrm{CONT}(a, C, A)$ holds (we say that the run of the program can continue from a to A through C) then $\forall m \in \gamma_{\mathfrak{m}}(a).\ \exists m' \in \gamma_{\mathfrak{m}}(A).\ (m, m') \in T_{\mathbb{M}}(C)$. We define CONT separately for different kinds of atomic statements. In this paper, we consider numeric programs, which, apart from passive and assumption statements, can use:

(i) a *deterministic assignment* $x = expr$, which assigns the value of an expression *expr* to a program *variable* x;

(ii) a *nondeterministic assignment*, or *forget operation*, $x = *$, which assigns a non-deterministically selected value to a program variable x.

For the memory abstract domain, let us introduce an additional *coverage* operation $\sqsubseteq^+_{\mathfrak{m}}$ that generalizes abstract order. For an abstract memory state $a \in \mathbb{D}_{\mathfrak{m}}$ and a set $A \subseteq \mathbb{D}_{\mathfrak{m}}$, it should be that if $a \sqsubseteq^+_{\mathfrak{m}} A$ (we say that a is *covered* by A) then $\gamma_{\mathfrak{m}}(a) \subseteq \gamma_{\mathfrak{m}}(A)$. For an arbitrary domain, coverage can be defined via Hoare order: $a \sqsubseteq^+_{\mathfrak{m}} A$ iff $\exists a' \in A.\ a \sqsubseteq_{\mathfrak{m}} a'$. For a numeric domain, it is usually possible to define a more precise coverage operation. For example, the Parma Polyhedra Library [6] defines a specialized coverage operation for finite sets of convex polyhedra.

We define CONT as follows. For $a \in \mathbb{D}_{\mathfrak{m}}, A \subseteq \mathbb{D}_{\mathfrak{m}}$,

(i) For the passive statement *skip*, $\mathrm{CONT}(a, \mathit{skip}, A) \equiv a \sqsubseteq^+_{\mathfrak{m}} A$. Indeed, if $a \sqsubseteq^+_{\mathfrak{m}} A$ then $\gamma_{\mathfrak{m}}(a) \subseteq \gamma_{\mathfrak{m}}(A)$, and hence $\forall m \in \gamma_{\mathfrak{m}}(a).\ \exists m' = m \in \gamma_{\mathfrak{m}}(A).\ (m, m') = (m, m) \in T_{\mathbb{M}}(\mathit{skip})$.

(ii) For an assumption statement $[\theta]$, $\mathrm{CONT}(a, [\theta], A) \equiv \mathrm{eval}^{\mathfrak{m}}(\theta, a) = 1 \wedge a \sqsubseteq^+_{\mathfrak{m}} A$. Indeed, if $\mathrm{eval}^{\mathfrak{m}}(\theta, a) = 1$, then $\gamma_{\mathfrak{m}}(a) \subseteq [\![\theta]\!]$, and if additionally $a \sqsubseteq^+_{\mathfrak{m}} A$ then $\forall m \in \gamma_{\mathfrak{m}}(a).\ \exists m' = m \in \gamma_{\mathfrak{m}}(A).\ (m, m') = (m, m) \in T_{\mathbb{M}}([\theta])$.

(iii) For a nondeterministic assignment $x = *$, we use the fact that in many numeric domains (including the polyhedral domain) the pre-condition of $x = *$ can be computed precisely (via [24] *cylindrification* operation). That is, for $a \in \mathbb{D}_{\mathfrak{m}}$, $\gamma_{\mathfrak{m}}(\mathrm{pre}^{\mathfrak{m}}(x = *, a)) = \{m \in \mathbb{M} \mid \exists m' \in \gamma_{\mathfrak{m}}(a).\ (m, m') \in T_{\mathbb{M}}(x = *)\}$. In this case, $\mathrm{CONT}(a, x = *, A) \equiv a \sqsubseteq^{+}_{\mathfrak{m}} \{\mathrm{pre}^{\mathfrak{m}}(x = *, a') \mid a' \in A\}$.

(iv) Finally, for every other atomic statement C with left-total input-output relation $T_{\mathbb{M}}(C)$ (e.g., a deterministic assignment), $\mathrm{CONT}(a, C, A) \equiv \mathrm{post}^{\mathfrak{m}}(C, a) \sqsubseteq^{+}_{\mathfrak{m}} A$. Indeed, in this case $\gamma_{\mathfrak{m}}(A) \supseteq \gamma_{\mathfrak{m}}(\mathrm{post}^{\mathfrak{m}}(C, a)) \supseteq \mathrm{post}(C, \gamma_{\mathfrak{m}}(a))$. Since additionally, $T_{\mathbb{M}}(C)$ is left-total then for every $m \in \gamma_{\mathfrak{m}}(a).\ \exists m' \in \gamma_{\mathfrak{m}}(A).\ (m, m') \in T_{\mathbb{M}}(C)$.

Another way to look at it is that (iv) represents a general case that allows handling atomic statements with left-total input-output relations. Then, we specialize CONT for non-deterministic statements and for statements with non-left-total input-output relations. Case (iii) specializes CONT for non-deterministic assignments. It allows us to detect a situation where there exists a *specific* non-deterministic choice (i.e., a specific new value of a variable) that keeps the execution inside the recurrent set. Case (ii) specializes CONT for assumption statements (with non-left-total input-output relations). By extending the definition of CONT, we can extend our analysis to support more kinds of atomic statements. Note that the predicate CONT is defined using operations that are standard in program analysis.

Theorem 1. *Let $R_s \in \mathbb{D}_{\mathfrak{l}\sharp}$ be an element of $\mathbb{D}_{\mathfrak{l}\sharp}$ and $\mathbb{P}_s$ be a sub-program. Let it be that for every location $l \in \mathbb{L}_s$, abstract path $q \in \mathbb{D}_{\mathfrak{p}}$, and an abstract memory state $a \in \mathbb{D}_{\mathfrak{m}}$, s.t., $R_s(l)(q) = a$, there exists a successor location $l' \in \mathrm{succ}(l)|_{\mathbb{P}_s}$, s.t.* $\mathrm{CONT}(a, \mathfrak{c}(l, l'), \{a' \mid \exists q' \in \mathbb{D}_{\mathfrak{p}}.\ a' = R_s(l')(q')\})$. *Then, R_s represents a recurrent set of $\mathbb{P}_s$ and hence the whole program $\mathbb{P}$.*

Proof idea. The proof is a straightforward application of the definitions of CONT and $T_{\mathbb{S}}$. Intuitively, if $R_s \in \mathbb{D}_{\mathfrak{l}\sharp}$ satisfies the condition of the lemma, from every program state in $\alpha_{\mathfrak{s}}(\gamma_{\mathfrak{l}\sharp}(R_s))$ we can form a non-terminating semi-execution that only visits the elements of $\alpha_{\mathfrak{s}}(\gamma_{\mathfrak{l}\sharp}(R_s))$ – by executing the statements of $\mathbb{P}_s$ in a specific order. □

Thus, in the refinement step, we start with an element $W_s \in \mathbb{D}_{\mathfrak{l}\sharp}$ produced by the backward analysis, and from every location $l \in \mathbb{L}_s$, we repeatedly exclude the tuples $(q, a) \in W_s(l)$ that do not satisfy the condition of Theorem 1. Eventually, we arrive at an element $R_s \sqsubseteq_{\mathfrak{l}\sharp} W_s$ that satisfies Theorem 1 and hence, represents a recurrent set. Note that the refinement step that we implement in this paper is coarse. For some disjunct $(q, a) \in W_s(l)$, we either keep it unchanged or remove it as a whole. In particular, an empty set is trivially recurrent, and it is still sound to produce $R_s = \bot_{\mathfrak{l}\sharp}$. This is acceptable, as the main purpose of the refinement step is to ensure soundness, and the form of the recurrent set in our current implementation is inferred by the preceding backward and forward analyses. Although, the analysis would benefit from the ability to modify individual disjuncts during refinement (we leave this for future work).

Theorem 1 requires that for every location $l \in \mathbb{L}_s$ and abstract memory state $a = R_s(l)(q)$ (for some $q \in \mathbb{D}_{\mathfrak{p}}$), there is at least one edge $(l, l') \in \mathbb{E}_s$, s.t., for every program state $s \in \{(l, m) \in \mathbb{S} \mid m \in \gamma_{\mathfrak{m}}(a)\}$ there exists $s' \in \{(l', m') \mid \exists q \in \mathbb{D}_{\mathfrak{p}}.\ m' \in \gamma_{\mathfrak{m}}(R_s(l')(q))\}$, s.t. $(s, s') \in T_{\mathbb{S}}(\mathbb{P}_s)$. That is, for every abstract memory state in R_s, there exists at least one edge, s.t. taking this edge from any corresponding concrete state keeps the execution inside the recurrent set. This is viable in practice because of the choice of path domain $\mathbb{D}_\mathfrak{p}$ (which is described in the following section). Our path domain ensures that at every branching point, backward analysis always partitions the memory states by the branch that they are going to take at this branching point.

Finally, note that Theorem 1 can be used to find a recurrent set of the whole program (not necessarily a strongly connected sub-program $\mathbb{P}_s$) and this way, prove non-termination. If $\gamma_\sharp(R_s(\mathfrak{l}_\vdash)) \neq \varnothing$, then there exists at least one non-terminating program execution (a non-terminating semi-execution starting in the initial location). Unfortunately, so far, we have not had practical success with this approach. Our path domain $\mathbb{D}_\mathfrak{p}$, while sufficient to capture non-terminating paths through loops (esp., non-nested loops), is not precise enough to capture non-terminating paths through the whole program. Thus, for practical reasons, we search for recurrent sets of individual loops and assume that reachability analysis will be used to complete the non-termination proof.

3.3 Path Domain

For the path domain, in this paper, we use finite sequences of *future branching choices*. A *branching point* is a location $l \in \mathbb{L}$, s.t., there exists at least two edges from l. A *branching choice* is an edge $(l, l') \in \mathbb{E}$, s.t., l is a branching point. We denote the set of all branching choices by $\mathbb{E}_\mathfrak{b} \subseteq \mathbb{E}$. For every non-bottom element $q \in \mathbb{D}_\mathfrak{p}$, q is a finite sequence of branching choices: $q = \langle e_0, e_1, \ldots, e_n \rangle \in \mathbb{E}_\mathfrak{b}^*$; top element $\top_\mathfrak{p}$ is the empty sequence $\langle\rangle$; and bottom is a distinguished element $\bot_\mathfrak{p} \notin \mathbb{E}_\mathfrak{b}^*$. E.g., for our running example in Fig. 1, l_2 is a branching point, and the branching choices are (l_2, l_3) and (l_2, l_4) For $q_1, q_2 \in \mathbb{D}_\mathfrak{p}$, $q_1 \sqsubseteq_\mathfrak{p} q_2$ if $q_1 = \bot_\mathfrak{p}$ or q_2 is a prefix of q_1. For $q_1, q_2 \in \mathbb{D}_\mathfrak{p}$, join $q_1 \sqcup_\mathfrak{p} q_2$ is q_2 if $q_1 = \bot_\mathfrak{p}$, q_1 if $q_2 = \bot_\mathfrak{p}$, or the longest common prefix of q_1 and q_2 otherwise. For $q_1, q_2 \in \mathbb{D}_\mathfrak{p}$, meet $q_1 \sqcap_\mathfrak{p} q_2 = q_1$ if $q_1 \sqsubseteq_\mathfrak{p} q_2$, q_2 if $q_2 \sqsubseteq_\mathfrak{p} q_1$, and $\bot_\mathfrak{p}$ otherwise. Additionally, we require that every element $q \in \mathbb{D}_\mathfrak{p}$ is *bounded*, i.e., every branching choice $e \in \mathbb{E}_\mathfrak{b}$ appears in q at most k times for a parameter $k \geq 1$. For a sequence of branching choices $q' \in \mathbb{E}_\mathfrak{b}^*$ (or $\in \mathbb{E}_\mathfrak{b}^\mathbb{N}$), we can produce a bounded element $\mathfrak{b}_k(q') \in \mathbb{D}_\mathfrak{p}$ by keeping the longest bounded prefix of the sequence. An element $q = \langle e_0, e_1, \ldots, e_n \rangle \in \mathbb{E}_\mathfrak{b}^*$ represents the set of paths $\gamma_\mathfrak{p}(q) \subseteq \Pi$, s.t. $\pi = (\langle l_0, l_1, \ldots \rangle, i) \in \gamma_\mathfrak{p}(q)$ iff for $j = 0..n$, there exists a strictly increasing sequence of indices $\{x_j\} : i \leq x_0 < \ldots < x_n$, s.t., every $\pi_{(x_j)}$ is a branching point, $(\pi_{(x_j)}, \pi_{((x_j)+1)}) = e_j$, and for every index z, s.t. $i \leq z < x_n$, if $z \notin \{x_j\}$, then $\pi_{(z)}$ is not a branching point. Let us define a corresponding abstraction function. For a path $\pi = (\langle l_0, l_1, \ldots \rangle, i) \in \Pi$ and $j \geq 0$ let $\{y_j\}$ be a strictly increasing sequence of indices of branching points at or after position i: $i \leq y_0 < y_1 < \ldots$, every $\pi_{(y_j)}$ is a branching point, and for every index $z \geq i$, if $z \notin \{y_j\}$, then $\pi_{(z)}$ is not a branching point. Then, the abstraction of π is $\alpha_\mathfrak{p}(\pi) = \mathfrak{b}_k(\langle (\pi_{(y_0)}, \pi_{((y_0)+1)}), (\pi_{(y_1)}, \pi_{((y_1)+1)}), \ldots \rangle)$. For a set of paths V, $\alpha_\mathfrak{p}(V) = \bigsqcup_\mathfrak{p} \{\alpha_\mathfrak{p}(\pi) \mid \pi \in V\}$. For an edge $e \in \mathbb{E}$ and $q \in \mathbb{D}_\mathfrak{p}$, $\mathrm{pre}^\mathfrak{p}(e, q) = \bot_\mathfrak{p}$ if $q = \bot_\mathfrak{p}$; $\mathfrak{b}_k(e \cdot q)$ if $q \neq \bot_\mathfrak{p}$ and e is a branching choice; and q otherwise. Here $\cdot$ denotes concatenation. Respectively, $\mathrm{post}^\mathfrak{p}(e, q) = q'$ if $q = e \cdot q'$ for some $q' \in \mathbb{D}_\mathfrak{p}$; $\bot_\mathfrak{p}$ if $q = e' \cdot q'$ for some $q' \in \mathbb{D}_\mathfrak{p}$ and $e' \neq e$; $\top_\mathfrak{p}$ if $q = \top_\mathfrak{p}$; and $\bot_\mathfrak{p}$ if $q = \bot_\mathfrak{p}$.

Intuitively, an abstract path $q \in \mathbb{D}_\mathfrak{p}$ predicts a bounded number of branching choices that an execution would make. For our running example in Fig. 1, if we take $k = 1$ then the abstraction of the infinite path $\langle (l_1, l_2, l_3)^\mathbb{N} \rangle$ is $\langle (l_2, l_3) \rangle$. We observe that our path domain works well for non-nested loops, and the bound k is the number of loop iterations for which we keep the branching choices. In most our experiments, $k = 1$ or

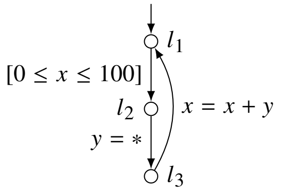

Fig. 2. Loop that assigns a non-deterministic value to a variable in every iteration.

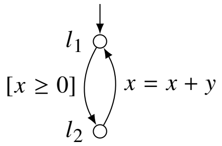

Fig. 3. Loop that requires a specific range of y for non-termination.

```
if (a < b)
   swap(a, b)
while (a ≠ b) {
   t = a − b;
   a = b;
   b = t;
}
```

Fig. 4. GCD algorithm with an introduced bug.

2 was enough to find a recurrent set. Note that the forward transformer $\mathrm{post}^{\mathfrak{p}}$ leaves $\top_{\mathfrak{p}}$ unchanged. Thus, our backward analysis does use trace partitioning, but the forward pre-analysis does not (with the current path domain). The forward pre-analysis, is initialized with $f_0 = \{\mathfrak{l}_{\vdash} \mapsto \top_{\sharp}; l \neq \mathfrak{l}_{\vdash} \mapsto \bot_{\sharp}\}$ where $\top_{\sharp} = \{\top_{\mathfrak{p}} \mapsto \top_{\mathfrak{m}}\}$, i.e., during the forward pre-analysis, every location is mapped either to $\bot_{\sharp}$ or to $\{\top_{\mathfrak{p}} \mapsto m\}$ for some $m \in \mathbb{D}_{\mathfrak{m}}$.

4 Examples of Handling Non-Determinism

In this section, we present numeric examples that demonstrate how different components of the analysis (trace partitioning, CONT, lower widening) are important for different kinds of non-terminating behaviors. In all examples, we assume that program variables are unbounded integers, and the analysis uses the polyhedral domain [19]. In Examples 1-3, we focus on a single loop and ignore that it can be a part of a larger program: e.g., we omit the branch that exits a loop, although it would usually be present in a program.

Example 1 – Non-deterministic branches For the program in Fig 1, a non-terminating execution in every iteration needs to make the choice depending on the current value of x, so that it does not go outside the range [0, 100]. This is captured by our path domain with $k = 1$ (the bound on the occurrences of the same branching choice in the abstract path). The first two steps (pre-analysis and backward analysis) yield the candidate recurrent set W_s. We do not describe these steps in detail, but $W_s(l_1) = \{\langle(l_2, l_3)\rangle \mapsto (0 \leq x \leq 99); \langle(l_2, l_4)\rangle \mapsto (1 \leq x \leq 100)\}$, $W_s(l_2) = W_s(l_1)$, $W_s(l_3) = \{\langle(l_2, l_3)\rangle \mapsto (1 \leq x \leq 99); \langle(l_2, l_4)\rangle \mapsto (1 \leq x \leq 100)\}$, and $W_s(l_4) = \{\langle(l_2, l_3)\rangle \mapsto (0 \leq x \leq 99); \langle(l_2, l_4)\rangle \mapsto (1 \leq x \leq 99)\}$. This can be interpreted as follows. If the execution is at location l_1 and, as the next branching choice, is going to increment x (by taking the edge (l_2, l_3)), then, for the execution to not leave the loop, it must be that $0 \leq x \leq 99$. Indeed, if $x < 0$, the execution will not enter the loop, and if $x > 99$, the execution will exit the loop after incrementing x. Similarly, if the execution is going to decrement x, it must be that $1 \leq x \leq 100$. That is, if the execution is at location l_1, and $0 \leq x \leq 100$, there exists a branching choice at location l_2 that keeps x in range [0, 100]. This way we can construct a non-terminating execution. Note that W_s represents a genuine recurrent set, and the final (refinement) step of the analysis yields $R_s = W_s$.

Example 2 – Non-deterministic assignment in the loop Fig. 2 shows a loop that in every iteration, first assigns a non-deterministic value to y and then adds it to x. Intuitively, if at location l_1 x is in range [0, 100], then for the edge (l_2, l_3), there is always a choice of y, s.t. x+y is still in the range [0, 100]. This way, we can construct a non-terminating execution. The way we specialize the predicate CONT to non-deterministic assignments allows us to handle such cases. The first two steps (pre-analysis and backward analysis) yield the candidate recurrent set W_s, s.t. $W_s(l_1) = \{\langle\rangle \mapsto (0 \leq x \leq 100)\}$, $W_s(l_2) = W_s(l_1)$, and $W_s(l_3) = \{\langle\rangle \mapsto (0 \leq x \leq 100 \wedge 0 \leq x+y \leq 100)\}$. We show that W_s satisfies Theorem 1 and thus represents a genuine recurrent set. Indeed. For location l_1, the successor location is l_2, and $(0 \leq x \leq 100)$ satisfies the memory-state formula of the assumption statement that labels (l_1, l_2). That is, for every state at location l_1 with $0 \leq x \leq 100$, we will stay in the recurrent set after executing the assumption statement. This corresponds to case (ii) of the predicate CONT. For location l_2, the successor location is l_3 and $\mathfrak{c}(l_2, l_3)$ is the non-deterministic assignment $y = *$. Note that for every value of x it is possible to choose a value of y, s.t. $0 \leq x+y \leq 100$ holds. Or, more formally, $\mathrm{pre}^{\mathfrak{m}}(y = *, (0 \leq x \leq 100 \wedge 0 \leq x+y \leq 100)) = (0 \leq x \leq 100)$ which corresponds to case (iii) of the predicate CONT. Finally, for location l_3, the successor location is l_1 and $\mathfrak{c}(l_3, l_1)$ is $x = x+y$. Also, $\mathrm{post}^{\mathfrak{m}}(x = x + y, (0 \leq x \leq 100 \wedge 0 \leq x+y \leq 100)) = (0 \leq x-y \leq 100 \wedge 0 \leq x \leq 100) \sqsubseteq (0 \leq x \leq 100)$ which corresponds to case (iv) of the predicate CONT. Therefore, W_s represents a genuine recurrent set, and the final step of the analysis yields $R_s = W_s$.

Example 3 – Non-deterministic assignment before the loop Fig. 3 shows a loop that in every iteration adds y to x. Both x and y are not initialized before the loop, and are thus assumed to take non-deterministic values. If at location l_1, $x \geq 0$ and $y \geq 0$, it is possible to continue the execution forever. Let us see how the constraint $y \geq 0$ can be inferred with lower widening. For this program, the pre-analysis produces the invariant F, s.t., $F(l_1) = \{\langle\rangle \mapsto \top\}$, and $F(l_2) = \{\langle\rangle \mapsto x \geq 0\}$. Then, consider a sequence of approximants $\{w_i\}_{i \geq 0}$ where $w_0 = F$ and for $i \geq 1$, $w_i = w_{i-1} \sqcap_{\mathfrak{I}^\sharp} \mathrm{pre}^{\mathfrak{I}^\sharp}(\mathbb{P}, w_{i-1})$ which corresponds to running the backward analysis without lower widening. Then, we will observe that the i-th approximant at location l_1 represents the condition that ensures that the execution will make at least i iterations through the loop. For $i \geq 0$, let $w'_i = w_i(l_1)(\langle\rangle)$. Then, $w'_0 = \top$, $w'_1 = x \geq 0$, $w'_2 = (x \geq 0 \wedge x+y \geq 0)$, $w'_3 = (x \geq 0 \wedge x+2y \geq 0)$, $w'_4 = (x \geq 0 \wedge x+3y \geq 0)$, and so on. That is, for $i \geq 1$, $w'_i = (x \geq 0 \wedge x + iy \geq 0)$ (a polyhedron with a "rotating" constraint), and we would like a lower widening technique that would produce an extrapolated polyhedron $(x \geq 0 \wedge y \geq 0)$ which is the limit of the chain $\{w'_i\}_{i \geq 0}$. Notice how this limit is below w'_i for every $i \geq 0$. This explains why we use lower widening (and not, e.g., narrowing) to ensure convergence of the backward analysis. Here, we use lower widening as proposed by A. Miné [30]. Intuitively, it works by retaining stable generators (which can be seen as dual to standard widening that retains stable constraints). Additionally, we use widening delay of 2 and a technique of threshold rays (also described in [30]), adding the coordinate vectors and their negations to the set of thresholds. Alternatively, instead of using threshold rays, one could adapt to lower widening the technique of evolving rays [7]. This allows the backward analysis to produce the extrapolated polyhedron $(x \geq 0 \wedge y \geq 0)$. Eventually, backward analysis produces the candidate W_s where

$W_s(l_1) = \{\langle\rangle \mapsto (x \geq 0 \wedge y \geq 0)\}$ and $W_s(l_2) = W_s(l_1)$. W_s represents a genuine recurrent set, and the final (refinement) step of the analysis yields $R_s = W_s$.

Example 4 This example is a program "GCD" from the test set [3] of Invel [35]. The program given in pseudocode in Fig. 4 is based on the basic algorithm that computes the greatest common divisor of two numbers: a and b – but has an introduced bug that produces non-terminating behaviors. For the loop in this program, our analysis (with $k = 2$) is able to show that if at line 3, it is the case that $(a > b \wedge a > 2b)$ or $(b > a \wedge 2b > a)$, the execution will never terminate and will alternate between these two regions. This example demonstrates how the interaction between the components of the analysis allows finding non-trivial non-terminating behaviors. In a program graph, the condition $a \neq b$ will be represented by a pair of edges, labelled by assumption statements: $[a > b]$ and $[a < b]$. Thus, these assumption statements become branching choices at line 3. Then, the path domain (with k at least 2) allows the analysis to distinguish the executions that alternate between these two assumption statements for the first k loop iterations. By doing numeric reasoning, one can check that there exist non-terminating executions that alternate between the two assumption statements indefinitely.

The example also demonstrates a non-trivial refinement step. At line 3, backwards analysis actually yields two additional disjuncts, one of those being $(a > b \wedge 2b > a \wedge 3b - a > 4)$. These are the states that take the branching choice $[a > b]$ for at least two first loop iterations. But from some of the concrete states in the disjunct, e.g., $(a = 6, b = 4)$, the loop eventually terminates. As currently implemented, the refinement step has to remove the whole disjunct from the final result.

Finally, note how for this example, recurrent set cannot be represented by a single convex polyhedron (per program location). Our approach allows to keep multiple polyhedra per location, corresponding to different abstract paths.

To summarize, the components of the analysis are responsible for handling different features of non-terminating executions. Trace partitioning allows predicting paths that non-terminating executions take; predicate CONT deals with non-deterministic statements in a loop; lower widening infers the required values of variables that are non-deterministically set outside of a loop.

5 Experiments

Our prototype implementation supports numeric programs (with some restrictions) and uses the product of polyhedra and congruences (via Parma Polyhedra Library [6]) as the memory domain. We applied our tool to the test set [3] of Invel [35], to non-terminating programs from SV-COMP 2015 [4] termination category (manually converted to our

		This paper				AProVE			Automizer		HipTNT+		SAS15		
	Tot.	OK	M	U	X	OK	?	X	OK	X	OK	X	OK	?	X
Invel	53	46	5	2	-	51	-	2	-	-	-	-	39(+1)	1	13(+12)
SVCOMP	44	32	-	9	3	30(+6)	4	10(+6)	37(+11)	7(+6)	35(+7)	9(+4)	-	-	-

Table 1. Experimental results

tool's input language), and additionally, to a set of non-deterministic numeric programs that we produced ourselves (all test programs are non-terminating, i.e., every program has at least one non-terminating behavior). Table 1 summarizes the results for the Invel and SV-COMP non-terminating programs and compares our tool to 3 existing tools: AProVE [20], Automizer [22], and HipTNT+ [27], and additionally to the authors' previous work on finding universal recurrent sets with forward analysis [8], column "SAS15". For Automizer and HipTNT+, we do not have the results for Invel programs, and for [8], there are no results for SV-COMP benchmarks. For AProVE, we give results for Invel programs as reported by [12] and for SV-COMP programs, as reported by the Non-Termination competition 2015 [5] (the version of AProVE that participated in SV-COMP did not include a non-termination prover for C programs). For our tool, column "OK" is the number of programs for which our tool finds a recurrent set. In most cases $k = 1$ or 2 was enough to find a recurrent set. In some cases, we need $k = 4$. The sets were later checked *manually* for reachability. Most test programs consist of a single non-terminating loop and a stem that gives initial values to program variables; and to check reachability, we only needed to intersect the inferred recurrent set with the produced set of initial states. Column"M" is the number of programs that originally fall outside of the class that our tool can handle, but after we introduced small modifications (e.g., replaced a non-linear condition with an equivalent linear one), our tool finds a recurrent set for them. Column "U" is the number of programs for which no recurrent set could be found due to technical limitations of our tool that does not support arrays, pointers, some instances of modular arithmetic, etc. Column "X" is the number of programs for which no recurrent set could be found for other reasons. This is usually due to overly aggressive lower widening, which could be improved in the future by introducing relevant widening heuristics. Note that our tool always terminates, i.e., failure means that it produced an empty recurrent set. For the other tools, the columns "OK" and "X" give the number of programs for which the tools were able, and respectively failed to prove non-termination. In brackets, we give the number of programs for which our tool gives the opposite outcome. Column "?" gives the number of programs for which we did not find reported results.

Table 1 should not be interpreted as a *direct* comparison of our tool or approach with the other tools. On one hand, our results are not subsumed by other tools, and we were able to find recurrent sets for some programs, where other tools failed to prove non-termination. On the other hand, the tools prove different things about the programs. Our tool finds recurrent sets of loops; AProVE and Automizer prove the existence of at least one non-terminating execution; the analysis of [8] and HipTNT+ (to our knowledge) prove that from some initial states, all executions are non-terminating. Also, the analysis of [8] is not optimized for numeric programs: e.g., it uses interval domain, while the analysis that we present uses the more expressive polyhedral domain.

6 Related Work

The idea of proving non-termination by looking at paths of a certain form appears in multiple existing approaches. An early analysis by Gupta et al. [21] produces proofs of non-termination from lasso-shaped symbolic executions using Farkas' lemma. Automizer [22, 23] decomposes the original program into a set of lasso-programs (a stem and a loop with no branches) to separately infer termination or non-termination [28] arguments for them. AProVE [20] implements a range of techniques. One of those [12], from a set of paths through a loop, produces a formula that is unsatisfiable if there is a set of states that cannot be escaped by following these paths. In a similar way, our approach uses trace partitioning to identify a path through a loop that a non-terminating execution takes. This does not have to be the same path segment repeated infinitely often, but may be an alternation of different segments. We see a strength of our approach in that it is parameterized by a path domain. That is, the partitioning scheme can be improved in future work and/or specialized for different classes of programs.

Chen et al. [13] use a combination of forward and backward analysis, but in a different way. With forward analysis, they identify terminating abstract traces; then using backward analysis over a single trace, they restrict the program (by adding assumption statements) to remove this trace. In contrast, our approach uses backward analysis to produce a candidate recurrent set, by computing an approximation of its fixed point characterization. Then, they show that the restricted program has at least one execution (non-terminating by construction). This is similar to the final step of our analysis.

A number of approaches prove that from some input states, a program does not have terminating behaviors (in contrast to proving the existence of at least one non-terminating behavior). That is, they find a set of states from which a program cannot escape. This can be done using Farkas' lemma [14], forward [8] or backward [34] abstract interpretation based analysis, or by encoding the search as a max-SMT problem [26]. Le et al. propose a specification logic and an inference algorithm [27] (implemented in HipTNT+) that can capture the absence of terminating behaviors. Invel [35] uses a template and a refinement scheme to infer invariants proving that final states of a program are unreachable.

A distinctive approach implemented in E-HSF [10] allows to specifying the semantics of programs and expressing verified properties (including the existence of different kinds of recurrent sets) in the form of ∀∃ quantified Horn clauses.

Finally, [29] presents a different formalization of trace partitioning (in the context of standard forward analysis), and [18] – of trace semantics.

7 Conclusion and Future Work

We proposed an analysis that finds existential recurrent sets of the loops in imperative programs. The analysis is based on the combination of forward and backward abstract interpretation and an important technique that we use is trace partitioning. To our knowledge, this is the first application of trace partitioning to backward analysis. The implementation of our approach for numeric programs demonstrated results that are comparable to those of state-of-the-art tools. As directions of future work we see: first, to develop a more precise path domain. Having a domain that can represent, e.g., lasso-shaped paths would allow better handling of nested loops and extending our technique to proving non-termination (rather than finding recurrent sets). Second, to extend our prototype to support additional memory domains (e.g., for shape analysis). Finally, the analysis of numeric programs will benefit from a specialized numeric refinement step.

Acknowledgements A. Bakhirkin is supported by a Microsoft Research Scholarship.

References

1. http://www.zuneboards.com/forums/showthread.php?t=38143, last accessed in October 2015.
2. http://azure.microsoft.com/blog/2014/11/19/update-on-azure-storage-service-interruption, last accessed in October 2015.
3. http://www.key-project.org/nonTermination/, last accessed in October 2015.
4. http://sv-comp.sosy-lab.org/2015/, last accessed in October 2015.
5. http://www.termination-portal.org/wiki/Termination_Competition_2015, last accessed in October 2015.
6. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1–2), 3–21 (2008)
7. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise widening operators for convex polyhedra. Sci. Comput. Program. 58(1-2), 28–56 (2005)
8. Bakhirkin, A., Berdine, J., Piterman, N.: A forward analysis for recurrent sets. In: Blazy, S., Jensen, T. (eds.) SAS. LNCS, vol. 9291, pp. 293–311. Springer (2015)
9. Berdine, J., Bjørner, N., Ishtiaq, S., Kriener, J.E., Wintersteiger, C.M.: Resourceful reachability as HORN-LA. In: McMillan, K.L., Middeldorp, A., Voronkov, A. (eds.) LPAR. Lecture Notes in Computer Science, vol. 8312, pp. 137–146. Springer (2013)
10. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified Horn clauses. In: Sharygina, N., Veith, H. (eds.) CAV. LNCS, vol. 8044, pp. 869–882. Springer (2013)
11. Biere, A., Bloem, R. (eds.): Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, LNCS, vol. 8559. Springer (2014)
12. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and nullpointerexceptions for java bytecode. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS. LNCS, vol. 7421, pp. 123–141. Springer (2011)
13. Chen, H.Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.W.: Proving nontermination via safety. In: Ábrahám, E., Havelund, K. (eds.) TACAS. LNCS, vol. 8413, pp. 156–171. Springer (2014)
14. Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.W.: Disproving termination with overapproximation. In: FMCAD. pp. 67–74. IEEE (2014)
15. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. Commun. ACM 54(5), 88–98 (2011)
16. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: Piterman, N., Smolka, S.A. (eds.) TACAS. LNCS, vol. 7795, pp. 47–61. Springer (2013)
17. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Automated Software Engineering 6(1), 69–95 (1999)
18. Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: Field, J., Hicks, M. (eds.) POPL. pp. 245–258. ACM (2012), http://dl.acm.org/citation.cfm?id=2103656
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) POPL. pp. 84–96. ACM Press (1978)
20. Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Proving termination of programs automatically with aprove. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR. LNCS, vol. 8562, pp. 184–191. Springer (2014)
21. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: Necula, G.C., Wadler, P. (eds.) POPL. pp. 147–158. ACM (2008)

22. Heizmann, M., Dietsch, D., Leike, J., Musa, B., Podelski, A.: Ultimate automizer with array interpolation - (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS. LNCS, vol. 9035, pp. 455–457. Springer (2015)
23. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere and Bloem [11], pp. 797–813
24. Henkin, L., Monk, J.D., Tarski, A.: Cylindric Algebras: Part I. North-Holland (1971)
25. Kleene, S.: Introduction to Metamathematics. North-Holland, second edn. (1987)
26. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using max-smt. In: Biere and Bloem [11], pp. 779–796
27. Le, T.C., Qin, S., Chin, W.: Termination and non-termination specification inference. In: Grove, D., Blackburn, S. (eds.) PLDI. pp. 489–498. ACM (2015)
28. Leike, J., Heizmann, M.: Geometric series as nontermination arguments for linear lasso programs. CoRR abs/1405.4413 (2014)
29. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, S. (ed.) ESOP. LNCS, vol. 3444, pp. 5–20. Springer (2005)
30. Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. Science of Computer Programming p. 33 (Oct 2013)
31. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
32. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (1972)
33. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings. LNCS, vol. 8723, pp. 302–318. Springer (2014)
34. Urban, C., Miné, A.: Proving guarantee and recurrence temporal properties by abstract interpretation. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings. Lecture Notes in Computer Science, vol. 8931, pp. 190–208. Springer (2015), http://dx.doi.org/10.1007/978-3-662-46081-8_11
35. Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In: Beckert, B., Hähnle, R. (eds.) TAP. LNCS, vol. 4966, pp. 154–170. Springer (2008)