# Abstraction Refinement for Termination*

Byron Cook[1], Andreas Podelski[2], and Andrey Rybalchenko[2]

[1] Microsoft Research, Cambridge
[2] Max-Planck-Institut für Informatik, Saarbrücken

**Abstract.** Abstraction can often lead to spurious counterexamples. Counterexample-guided abstraction refinement is a method of strengthening abstractions based on the analysis of these spurious counterexamples. For invariance properties, a counterexample is a finite trace that violates the invariant; it is spurious if it is possible in the abstraction but not in the original system. When proving termination or other liveness properties of infinite-state systems, a useful notion of spurious counterexamples has remained an open problem. No counterexample-guided abstraction refinement algorithm was known for termination. In this paper, we address this problem and present the first known automatic counterexample-guided abstraction refinement algorithm for termination proofs. We exploit recent results on transition invariants and transition predicate abstraction. We identify two reasons for spuriousness: abstractions that are too coarse, and candidate transition invariants that are too strong. Our counterexample-guided abstraction refinement algorithm successively weakens candidate transition invariants and refines the abstraction.

## 1 Introduction

The correctness argument for a program can often be based on a small fraction of the original program code. However, it is hard to extract this core automatically if the program is large and complex.

Automated abstraction refinement [6, 19] is designed to solve precisely this problem. It automatically extracts just the information that is needed to prove the correctness property. Such algorithms are known for safety and invariance properties [2, 5, 6, 13–17, 19]. However, no such algorithm is known for termination proofs of infinite-state systems.

Abstraction refinement is based on the notion of spurious counterexamples. For invariance properties, a counterexample is a finite trace that violates the invariant. The counterexample is spurious if the trace is possible in the abstract system, but infeasible in the concrete system. The proof of the infeasibility of

* The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

the trace provides guidance for adding more precision to the abstraction (and thus refining it).

For termination and liveness properties of infinite-state programs, a useful notion of spurious counterexamples has been an open problem. In this paper, we address this problem and present the first known counterexample-guided abstraction refinement algorithm for termination.

We follow a recent approach to temporal verification of infinite-state systems that is based on transition invariants [21] and transition predicate abstraction [22]. This approach is a promising starting point for the development of our refinement method because of its connection with abstraction methods [11]. Let $T$ be the transition relation of the infinite-state system. Transition invariant is the least fixed point of an operator $\mathcal{F}$ (defined as $\mathcal{F}(Q) = Q \circ T$), or rather its abstraction wrt. a set of transition predicates. The least fixed point construction is in analogy with abstract proofs for invariance properties, but the analogy stops here. Let us explain this point in detail.

Let $I$ be an invariant and $F$ be an operator such that

$$F(X) = \{s' \mid s \in X \text{ and } (s, s') \in T\} \ .$$

To prove that the invariant $I$ holds we can search for an abstraction $F^{\#}_{P}$ based on a set of predicates $P$ such that the least fixed point of $F^{\#}_{P}$ is contained in $I$:

$$\exists P. \quad \mathsf{lfp}(F^{\#}_{P}) \subseteq I \ .$$

The termination property does not come with an a priori fixed transition invariant. Any transition invariant is sufficient. Again, let $\mathcal{F}$ be $\mathcal{F}(Q) = Q \circ T$. In addition to finding an abstraction $\mathcal{F}^{\#}_{\mathcal{P}}$ of $\mathcal{F}$, we need to find a transition invariant $\mathcal{R}$ such that the least fixed point of $\mathcal{F}^{\#}_{\mathcal{P}}$ is contained in $\mathcal{R}$:

$$\exists \mathcal{R}\ \exists \mathcal{P}. \quad \mathsf{lfp}(\mathcal{F}^{\#}_{\mathcal{P}}) \subseteq \mathcal{R} \ . \tag{1}$$

The existence of the transition invariant $\mathcal{R}$ implies termination if $\mathcal{R}$ satisfies an additional property that we will explain later.

Thus, an automated termination checker that implements counterexample-guided abstraction refinement must not only construct an appropriate set of transition predicates $\mathcal{P}$, but also an assertion $\mathcal{R}$ that is an appropriate transition invariant. When the inclusion (1) does not hold, we do not know whether the left side is too big or the right side is not big enough. Thus, our refinement algorithm analyzes the reason why $\mathsf{lfp}(\mathcal{F}^{\#}_{\mathcal{P}})$ is not included in $\mathcal{R}$. Then, it chooses accordingly one of two possible actions. Either it decides that the abstraction is too coarse and it refines the abstraction by adding more transition predicates to $\mathcal{P}$ and thus makes $\mathsf{lfp}(\mathcal{F}^{\#}_{\mathcal{P}})$ smaller, or it decides that the candidate transition invariant $\mathcal{R}$ is too strong and weakens it.

This leads to a notion of counterexample that reflects both aspects of spuriousness: A counterexample is spurious if either the abstraction is too coarse or the candidate transition invariant is too strong. It is this new notion of spurious counterexamples that leads to the first known counterexample-guided abstraction refinement for the automation of termination proofs.

## 2 Related work

Our work builds upon and benefits from the previous research on abstraction refinement (e.g. [2, 5, 6, 13–17, 19]) and automatic termination proofs (e.g [4, 8, 10, 12, 20]) for infinite-state systems. A short way to distinguish our work from the existing literature in those two research areas is that we are the first to discover a method of abstraction refinement for termination analysis of infinite-state systems.

For the comparison with existing abstraction refinement tools: none of those tools can automatically prove termination, except for in trivial cases. This limitation is inherent to *predicate* abstraction (see [22] for an explanation).

The approach in [1] is to encode ranking functions into fairness assumptions for a finite model obtained by predicate abstraction; in contrast with our work, the actual termination arguments are ranking functions (which are found manually or by the above-mentioned tools without abstraction refinement).

The work in [22] presents an algorithm that, for a given set of *transition predicates*, constructs an abstraction of a program for the verification of liveness properties. This work does not, however, provide any guidance on how to refine the abstraction if it fails to prove the property.

Other proof methods for liveness properties have been proposed that are limited to only finite-state systems. For example the work in [3] exploits the fact that a non-terminating finite-state system must visit the same state infinitely many times.

## 3 Preliminaries

**Programs** Following [18], we abstract away from the syntax of a concrete programming language such as C and represent a program $P$ by a set of transitions. Each transition $\tau$ (to be thought of as the label of a program statement) refers to a transition constraint $\rho_\tau$, which is an assertion over the program variables and their primed versions.

We use $V$ and $V'$ to represent the set of variables of the program and the set of their primed versions, respectively. The intended semantics of $V'$ is to refer to the values of the variables $V$ after executing a transition. The set $V$ includes the variable $\mathsf{pc}$ (the program counter) which ranges over the program *locations*.

Each transition $\tau$ refers to a pair $(\ell, \ell')$ of *pre* and *post* locations, respectively. These locations appear in the transition constraint $\rho_\tau$ in the form of the conjuncts $\mathsf{pc} = \ell$ and $\mathsf{pc}' = \ell'$. The program has an *initial location* $\ell^0$ and an *initial condition* $\Theta$, which is an assertion over program variables. The initial location $\ell^0$ appears in $\Theta$ as the conjunct of the form $\mathsf{pc} = \ell^0$.

We assume that the program $P$ is fixed from now on.

**Program Semantics** A program *state* $s$ is a valuation of the program variables, including the program counter $\mathsf{pc}$.

We identify an *assertion over program variables* with the set of *states that it denotes*. For example, $\Theta$ is the the set of initial states. We also identify an *assertion over primed and unprimed program variables* with the *set of pairs of states that it denotes*. For example, $\rho_\tau$ is the transition relation of the transition $\tau$.

A *computation* $s_0, s_1, s_2, \ldots$ is a possibly infinite sequence of states that starts in an initial state ($s_0 \in \Theta$) and that is consecutive, *i.e.*, each pair of successive states belongs to the transition relation of some transition. Formally, for each $i \geq 0$ there exists a transition $\tau$ such that $(s_i, s_{i+1}) \in \rho_\tau$.

**Paths and Cyclic Paths** A *path* $\pi = \tau_1 \ldots \tau_n$ is a (finite) sequence of transitions with consecutive locations (the post location of $\tau_i$ is the pre location of $\tau_{i+1}$). A *cyclic path* $\pi = \tau_1 \ldots \tau_n$ is a special case of a path with the same start and end location (the pre location of its first transition $\tau_1$ is equal to the post location of its last transition $\tau_n$).[3]

We define the composition of relations $\rho_1 \circ \rho_2$ as usual:

$$\rho_1 \circ \rho_2 \quad \equiv \quad \{(s, s') \mid (s, s'') \in \rho_1 \text{ and } (s'', s') \in \rho_2\} \ .$$

It can be implemented by logical operations over transition constraints.

A path $\pi$ denotes a transition relation $\rho_\pi$ that is naturally obtained by composing the transition relations of the transitions along the path. Formally, for a path $\pi = \tau_1 \ldots \tau_n$ we have:

$$\rho_\pi \quad \equiv \quad \rho_{\tau_1} \circ \cdots \circ \rho_{\tau_n} \ .$$

**Termination** A program is *terminating* if it does not admit any infinite computation. A binary relation $R$ is *well-founded* if there exists no infinite sequence $s_0, s_1, s_2, \ldots$ that is consecutive wrt. $R$ (formally, for each $i \geq 0$ we have $(s_i, s_{i+1}) \in R$).

The following fact is a consequence of Theorem 1 in [21] (by the fact that the transition relation of each path $\pi$ with different start and end locations $\ell$ and $\ell'$ is contained in the well-founded relation $R_{\ell,\ell'} \quad \equiv \quad \mathsf{pc} = \ell \ \wedge \ \mathsf{pc}' = \ell'$).

**Theorem 1 (Termination Condition [21]).** *The program is terminating if there exists a finite set of well-founded relations $\mathcal{R} = \{R_1, \ldots, R_m\}$ such that the transition relation $\rho_\pi$ of each cyclic path $\pi = \tau_1 \ldots \tau_n$ is included in one of the relations from $\mathcal{R}$.*

The termination condition in the theorem above is formally:

$$\text{for each cyclic path } \pi = \tau_1 \ldots \tau_n : \quad \rho_\pi \subseteq R_1 \text{ or } \ldots \text{ or } \rho_\pi \subseteq R_m \ . \tag{2}$$

[3] Note that a cyclic path with end location $\ell$ may have numerous other steps that pass through $\ell$.

**Transition Predicates** We use transition predicate abstraction [22] in order to obtain a termination condition that is stronger than (2), and that can be checked effectively. A *transition predicate* $p$ is an assertion over program variables and their primed version, *i.e.*, $p$ is a binary relation over states. In contrast, a (plain) predicate is an assertion over program variables, *i.e.*, a set of states. We use $\mathcal{P}$ to refer to a finite set of transition predicates. *Transition predicate abstraction* is similar to predicate abstraction if one replaces the set of program variables $V$ by the set $V \cup V'$.

An *abstraction function* $\alpha_{\mathcal{P}}$ maps a binary relation $\rho$ over states to a superset expressed by a conjunction of transition predicates. We assume that one can automatically construct the abstraction function $\alpha_{\mathcal{P}}$ for a given finite set of transition predicates $\mathcal{P}$. A possible definition is the abstraction of a relation $\rho$ by the conjunction of all transition predicates $p \in \mathcal{P}$ weaker than $\rho$ (and test the 'weaker-than' relation $\rho \models p$ with a theorem prover).

For our formal treatment in Theorem 3, we will use one basic fact about the abstraction function $\alpha_{\mathcal{P}}$: the abstraction of a relation $\rho$ is the relation itself (*i.e.* there is no loss of precision during abstraction) if $\rho$ can be expressed by a conjunction of transition predicates (see Theorem 13 in [9]). Formally,

$$\alpha_{\mathcal{P}}(\rho) = \rho \quad \text{if } \rho = p_1 \wedge \ldots \wedge p_n \quad \text{for } p_1, \ldots, p_n \in \mathcal{P} \ . \tag{3}$$

**Abstraction of Paths** We can construct an abstraction $\widehat{\alpha}_{\mathcal{P}}(\pi)$ for each path $\pi = \tau_1 \ldots \tau_n$ according to the following inductive definition.

$$\begin{aligned} \widehat{\alpha}_{\mathcal{P}}(\tau_1 \ldots \tau_n) &\equiv \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \rho) \quad \text{where } \rho = \widehat{\alpha}_{\mathcal{P}}(\tau_2 \ldots \tau_n) \\ \widehat{\alpha}_{\mathcal{P}}(\tau_n) &\equiv \alpha_{\mathcal{P}}(\rho_{\tau_n}) \end{aligned}$$

The abstraction of the path $\pi$ is always a superset of the transition relation of $\pi$, formally

$$\rho_\pi \subseteq \widehat{\alpha}_{\mathcal{P}}(\pi) \ .$$

We obtain a termination condition that is effective in the sense that one can compute an abstraction $\widehat{\alpha}_{\mathcal{P}}(\pi)$ of each possible (cyclic) path $\pi$.

**Theorem 2 (Termination Condition with Abstraction [22]).** *The program is terminating if there exists a finite set of well-founded relations $\mathcal{R} = \{R_1, \ldots, R_m\}$ such that* the abstraction $\alpha_{\mathcal{P}}(\pi)$ *of the transition relation of each cyclic path $\pi = \tau_1 \ldots \tau_n$ is included in one of the relations from $\mathcal{R}$.*

This 'effective' termination condition is formally:

$$\text{for each cyclic path } \pi = \tau_1 \ldots \tau_n : \ \widehat{\alpha}_{\mathcal{P}}(\pi) \subseteq R_1 \text{ or } \ldots \text{ or } \widehat{\alpha}_{\mathcal{P}}(\pi) \subseteq R_m \ . \tag{4}$$

For notational convenience, we overload the symbol $\alpha_{\mathcal{P}}$. We will use $\alpha_{\mathcal{P}}$ not only as a function on relations $\rho$, but also as a function $\widehat{\alpha}_{\mathcal{P}}$ over paths $\pi$. We need to

distinguish the two functions. The abstraction of the transition relation $\rho_\pi$ is in general a subset of the abstraction of the path $\pi$, formally,

$$\alpha_{\mathcal{P}}(\rho_\pi) \subseteq \alpha_{\mathcal{P}}(\pi) \ .$$

For example, given the transition relations

$$\begin{aligned} \rho_{\tau_1} &\equiv x' = x - 2, \\ \rho_{\tau_2} &\equiv x' = x + 1, \end{aligned}$$

and a singleton set of transition predicates

$$\mathcal{P} = \{x' \leq x\} \ ,$$

we have

$$\begin{aligned} \alpha_{\mathcal{P}}(\rho_{\tau_1 \tau_2}) &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \rho_{\tau_2}) \\ &= \alpha_{\mathcal{P}}(x' = x - 1) \\ &= x' \leq x \ , \end{aligned}$$

whereas

$$\begin{aligned} \alpha_{\mathcal{P}}(\tau_1 \tau_2) &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \alpha_{\mathcal{P}}(\tau_2)) \\ &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \mathsf{true}) \\ &= \alpha_{\mathcal{P}}(\mathsf{true}) \\ &= \mathsf{true} \ . \end{aligned}$$

Thus, we have $\alpha_{\mathcal{P}}(\rho_{\tau_1 \tau_2}) \subsetneq \alpha_{\mathcal{P}}(\tau_1 \tau_2)$.

## 4 Refinement for Termination

The termination condition (4) suggests that, given a set of well-founded relations $\mathcal{R} = \{R_1, \ldots, R_m\}$, the problem of refinement is to find the 'right' set of transition predicates $\mathcal{P}$. The set $\mathcal{P}$ is 'right' if the induced abstraction $\alpha_{\mathcal{P}}$ is sufficiently precise to infer an inclusion of the form $\alpha_{\mathcal{P}}(\pi) \subseteq R_j$ for every cyclic path $\pi$, see (4).

Our algorithm must, however, also find the 'right' set of well-founded relations $\mathcal{R} = \{R_1, \ldots, R_m\}$. The set $\mathcal{R}$ is 'right' if the inclusion $\rho_\pi \subseteq R_j$ holds 'in the concrete' for every path $\pi$, see (2).

**Counterexamples** Distinction between the two cases above complicates the notion of a spurious counterexample. Namely, if the abstract check (4) does not succeed for a cyclic path $\pi$, then this may be spurious for one of two reasons: either the set of transition predicates $\mathcal{P}$ was not yet 'right' or the set of well-founded relations $\mathcal{R}$ was not yet 'right'.

**Definition 1 (Spurious Counterexample).** *Given a set of transition predicates $\mathcal{P}$ and a set of well-founded relations $\mathcal{R} = \{R_1, \ldots, R_m\}$, a cyclic path $\pi = \tau_1 \ldots \tau_n$ is a* counterexample *wrt. $\mathcal{P}$ and $\mathcal{R}$ if its abstraction $\alpha_{\mathcal{P}}(\pi)$ is not contained in any relation in $\mathcal{R}$. Formally,*

$$\alpha_{\mathcal{P}}(\pi) \not\subseteq R_j \quad \textit{for each } j \in \{1, \ldots, m\} \ .$$

*The counterexample $\pi$ is* spurious *if either its relation $\rho_\pi$ is contained in some relation $R_j$ of $\mathcal{R}$ or its relation $\rho_\pi$ is well-founded. Formally,*

$$\rho_\pi \subseteq R_j \quad \textit{for some } j \in \{1, \ldots, m\} \quad \textit{or} \quad \rho_\pi \ \textit{well-founded.}$$

**The Algorithm** Figure 1 shows our counterexample-guided abstraction refinement for termination. For each new set of well-founded relations $\mathcal{R}$ and for each new set of transition predicates $\mathcal{P}$, the algorithm checks whether there exists a counterexample wrt. $\mathcal{R}$ and $\mathcal{P}$. It does so by going through all cyclic paths $\pi$ until no more new abstract values $\alpha_{\mathcal{P}}(\pi)$ can be computed. Although the number of cyclic paths is infinite, the search converges because the range of the abstraction function $\alpha_{\mathcal{P}}$ is finite (and determined by the number of transition predicates in $\mathcal{P}$).

If the algorithm finds no counterexample, it has succeeded in proving the termination property and it stops. If the algorithm finds a counterexample $\pi$, there are three possibilities.

1. The counterexample $\pi$ is spurious because the set of transition predicates was not yet 'right'. Formally, the inclusion between $\rho_\pi$ and some $R \in \mathcal{R}$ does not hold in the abstract, *i.e.* $\alpha_{\mathcal{P}}(\pi) \not\subseteq R$, but it does hold in the concrete, *i.e.* $\rho_\pi \subseteq R$. The refinement step adds a set of transition predicates $\mathcal{P}_{\text{path}}$ from the transition relation of every suffix of the path $\pi = \tau_1 \ldots \tau_n$ to the set $\mathcal{P}$. These predicates will eliminate this particular counterexample in the next iteration of the algorithm. The set of predicates $\mathcal{P}_{\text{loop}}$ guarantees that the refinement will not get 'stuck in a loop' discovering an infinite sequence of counterexamples $\pi$, $\pi\pi$, $\ldots$, $\pi^i$, $\ldots$. We will provide a formal statement describing the progress of refinement in Theorem 3.
2. The counterexample $\pi$ is spurious because the set of well-founded relations $\mathcal{R}$ was not yet 'right'. This means that for any $R \in \mathcal{R}$ the inclusion $\rho_\pi \subseteq R$ does not hold neither in the abstract nor in the concrete, but the transition relation $\rho_\pi$ of the cyclic path $\pi$ is well-founded. This means that the candidate set $\mathcal{R}$ is not yet 'right'. In that case a well-founded relation $R$ containing $\rho_\pi$ is added to $\mathcal{R}$. In the next iteration of the algorithm, the same counterexample $\pi$ may be found again, but then we will be in Case 1.
3. The counterexample $\pi$ is not spurious: the transition relation $\rho_\pi$ of the cyclic path $\pi$ is not well-founded. In that case, the algorithm has failed to prove the termination property and it stops. In this case $\pi^\omega$ may be a feasible infinite trace.

```
1   input
2      Program P
3   begin
4      𝓡 := ∅                                   (* set of well-founded relations *)
5      𝓟 := ∅                                   (* set of transition predicates *)
6      repeat
7         if exists π = τ_1 ... τ_n s.t. α_𝓟(π) ⊈ R for any R ∈ 𝓡 then
8            if exists R ∈ 𝓡 such that ρ_π ⊆ R then
9               (* refinement step *)
10              𝓟_path := ⋃_{i∈1..n} Preds(ρ_{τ_i} ∘ ··· ∘ ρ_{τ_n})
11              𝓟_loop := Preds(R) ∪ ⋃_{i∈1..n} Preds(ρ_{τ_i} ∘ ··· ∘ ρ_{τ_n} ∘ R)
12              𝓟 := 𝓟 ∪ 𝓟_path ∪ 𝓟_loop
13           else
14              if π is well-founded by the ranking relation R then
15                 (* weakening step *)
16                 𝓡 := 𝓡 ∪ {R}
17              else
18                 return "Counterexample cyclic path τ_1 ... τ_n"
19        else
20           return "Program P terminates"
21  end.
```

**Fig. 1.** Counterexample-guided abstraction refinement for termination. In line 7, we investigate abstractions $\alpha_{\mathcal{P}}(\pi)$ of cyclic paths by exploring the paths in a breadth-first way. The exploration converges since the range of the abstraction function $\alpha_{\mathcal{P}}$ is finite. In line 10, $\mathsf{Preds}(T)$ symbolically evaluates $T$ and then extracts the set of atomic formulas from the reduced expression.

**Well-Foundedness and Ranking Relations** A ranking function for a (terminating) program is defined by an expression $\mathsf{rank}$ over the program variables. Its value for each reachable program state is a non-negative integer that decreases during each computation step.

We write $\mathsf{rank}(V)$ for the expression in the program variables and $\mathsf{rank}(V')$ for the expression in the primed version of the program variables. A ranking function defined by the expression $\mathsf{rank}$ induces a well-founded relation (a *ranking relation*) $R$ in the following way:

$$R \equiv \mathsf{rank}(V) \geq 0 \wedge \mathsf{rank}(V') \leq \mathsf{rank}(V) - 1 \ .$$

We note the following observation.

*Remark 1.* A ranking relation $R$ is transitive. Formally,

$$R \circ R \subseteq R .$$

A cyclic path $\pi = \tau_1 \dots \tau_n$ defines a program fragment of a very specific form: it consists of one program location $\ell$ and one transition from $\ell$ to $\ell$ with the transition relation $\rho_\pi$. There exist several automatic methods and tools for the computation of ranking functions for such programs, *e.g.* [4, 7, 20, 24]. These tools can be used for implementing line 14 of the algorithm.

**Progress of Refinement** A newly detected *spurious* counterexample gives rise to a new refinement step and a new iteration of the algorithm. The refinement algorithm makes progress if for each newly detected *spurious* counterexample $\pi$ the cyclic path $\pi$ is no longer a counterexample after the next iteration or the next two iterations of the algorithm. Our algorithm enjoys the property of eliminating the infinite set of spurious counterexamples $\pi, \pi\pi, \dots$ in a single step. We formalize this property in Theorem 3.

**Theorem 3 (Progress of Refinement).** *If $\pi$ is a spurious counterexample wrt. the sets $\mathcal{R}$ and $\mathcal{P}$, then none of the cyclic paths $\pi_1, \pi_2, \dots$ obtained by concatenating $\pi$ with itself repeatedly ($\pi_1 = \pi$, $\pi_2 = \pi\pi$, etc.) is a counterexample wrt. the sets $\mathcal{R}'$ and $\mathcal{P}'$ obtained by refinement in one or possibly two more iterations of the algorithm in Figure 1.*

*Proof.* Given a spurious counterexample $\pi = \tau_1 \dots \tau_n$, there are two cases that we need to consider. In the first case, the relation $\rho_\pi$ is included in some $R \in \mathcal{R}$ (at line 8 on Figure 1). Hence, the refinement step (at lines 10, 11, and 12) updates the abstraction function. Now we consider the next iteration of the algorithm. Let $\mathcal{P}'$ be the current set of transition predicates, which define the abstraction function.

We prove that $\alpha_{\mathcal{P}'}(\pi^j) \subseteq R$ by induction over $j$.[4] For the base case $j = 1$, we prove $\alpha_{\mathcal{P}'}(\pi) \subseteq R$. By Theorem 13 in [9], an abstraction function is precise for some input if the input is expressible by the predicates defining the abstraction. Hence, for each $i \in \{1, \dots, n\}$ we have $\alpha_{\mathcal{P}'}(\tau_i \dots \tau_n) = \rho_{\tau_i} \circ \dots \circ \rho_{\tau_n}$. Thus, we have $\alpha_{\mathcal{P}'}(\pi) \subseteq R$.

For the induction step, we assume $\alpha_{\mathcal{P}'}(\pi^j) \subseteq R$ for some $j > 1$. By Theorem 13 in [9], we have $\alpha_{\mathcal{P}'}(\tau_i \dots \tau_n \pi^j) \subseteq \rho_{\tau_i} \circ \dots \circ \rho_{\tau_n} \circ R$ for each $i \in \{1, \dots, n\}$. Hence, we have $\alpha_{\mathcal{P}}(\pi\pi^j) = \rho_\pi \circ R$. Since $\rho_\pi \subseteq R$ and by the assumption that $R$ is a transitive relation, we have $\alpha_{\mathcal{P}}(\pi^{j+1}) \subseteq R \circ R \subseteq R$.

If $\rho_\pi$ is not contained in any $R \in \mathcal{R}$, then after the weakening step at line 16 using a ranking relation $R$ we have $\rho_\pi \subseteq R$, and the above case applies. □

## 5 Example

In this section we execute the algorithm contained in Figure 1 on a sample program fragment. Refer to left-hand side of Figure 2 for the example program.

[4] Note that we abstract wrt. a refined set of transition predicates $\mathcal{P}'$.

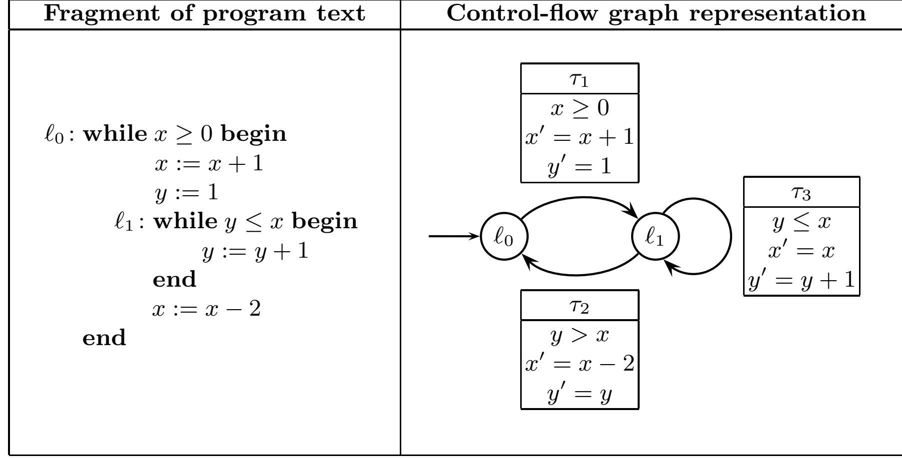


**Fig. 2.** Example program fragment with nested loops

We represent the program as a control-flow graph on the right-hand side, where each node is the start of a basic-block, and each transition (labeled $\tau_1$, $\tau_1$, and $\tau_3$) is decorated with a relation that represents the conditions and assignments of the basic block. We have the following transition relations $\rho_{\tau_i}$:

$$\begin{aligned}
\rho_{\tau_1} &\equiv x \geq 0 \wedge x' = x + 1 \wedge y' = 1 \wedge \mathsf{pc} = \ell_0 \wedge \mathsf{pc}' = \ell_1 \ , \\
\rho_{\tau_2} &\equiv y > x \wedge x' = x - 2 \wedge y' = y \wedge \mathsf{pc} = \ell_1 \wedge \mathsf{pc}' = \ell_0 \ , \\
\rho_{\tau_3} &\equiv y \leq x \wedge x' = x \wedge y' = y + 1 \wedge \mathsf{pc} = \ell_1 \wedge \mathsf{pc}' = \ell_1 \ .
\end{aligned}$$

To simplify the presentation, we assume an implicit treatment of the program counter. This means that we do not show any predicates involving $\mathsf{pc}$ in the exposition below.

We summarize the intermediate steps of our example execution in Table 1, and give a detailed explanation below. Line numbers refer to the algorithm shown on Figure 1.

**Step I/Line 4 and 5:** We start with the empty set of well-founded relations $\mathcal{R} = \emptyset$ and the empty set of transition predicates $\mathcal{P} = \emptyset$.

**Step II/Lines 7, 8, 10, 11, and 12:** We start enumerating cyclic paths and computing their abstractions. Because $\mathcal{R}$ is empty, we find that for the cyclic path $\pi = \tau_1\tau_2$ the abstract relation $\alpha_{\mathcal{P}}(\pi)$ does not entail any relations in $\mathcal{R}$. This means that $\pi$ is a counterexample. We do not know yet whether it is spurious. We therefore move to line 8. For the same reason there does not exist a relation $R$ in $\mathcal{R}$ such that $\rho_\pi \subseteq R$. We therefore move to line 14. The

| Step | Path $\pi$ | $\forall R \in \mathcal{R}$ | Action |
|---|---|---|---|
| **I** | - | - | Initialization with $\mathcal{R} = \emptyset$ and $\mathcal{P} = \emptyset$ |
| **II** | $\tau_1\tau_2$ | $\rho_\pi \not\subseteq R$ | Weakening with <br> $R_1 = \mathsf{false}$ |
| **III** | $\tau_1\tau_2$ | $\alpha_{\mathcal{P}}(\pi) \not\subseteq R$ | Refinement by <br> $\mathsf{Preds}(\rho_{\tau_2}) = \{y > x, x' = x - 2, y' = y\}$, <br> $\mathsf{Preds}(\rho_{\tau_1} \circ \rho_{\tau_2}) = \emptyset$, <br> $\mathsf{Preds}(\ldots R_1) = \emptyset$. |
| **IV** | $\tau_3$ | $\rho_\pi \not\subseteq R$ | Weakening with <br> $R_2 = x - y \geq 0 \wedge x' - y' \leq x - y - 1$ |
| **V** | $\tau_3$ | $\alpha_{\mathcal{P}}(\pi) \not\subseteq R$ | Refinement by <br> $\mathsf{Preds}(\rho_{\tau_3}) = \{y \leq x, x' = x, y' = y + 1\}$, <br> $\mathsf{Preds}(R_2) = \{x - y \geq 0, x' - y' \leq x - y - 1\}$, <br> $\mathsf{Preds}(\rho_{\tau_3} \circ R_2) = \{y \leq x - 1, x' - y' \leq x - y - 2\}$. |
| **VI** | $\tau_2\tau_1$ | $\rho_\pi \not\subseteq R$ | Weakening with <br> $R_3 = x \geq 2 \wedge x' \leq x - 1$ |
| **VII** | $\tau_2\tau_1$ | $\alpha_{\mathcal{P}}(\pi) \not\subseteq R$ | Refinement by <br> $\mathsf{Preds}(\rho_{\tau_1}) = \{x \geq 0, x' = x + 1, y' = 1\}$, <br> $\mathsf{Preds}(\rho_{\tau_2} \circ \rho_{\tau_1}) = \{y > x, x \geq 2, x' = x - 1, y' = 1\}$, <br> $\mathsf{Preds}(R_2) = \{x \geq 2, x' \leq x - 1\}$, <br> $\mathsf{Preds}(\rho_{\tau_1} \circ R_2) = \{x \geq 1, x' \leq x\}$, <br> $\mathsf{Preds}(\rho_{\tau_2} \circ \rho_{\tau_1} \circ R_2) = \{y > x, x \geq 3, x' \leq x - 3\}$. |
| **VIII** | $\tau_1\tau_3\tau_2$ | $\rho_\pi \not\subseteq R$ | Weakening with <br> $R_4 = x \geq 0 \wedge x' \leq x - 1$ |

**Table 1.** The states of the algorithm in Figure 1 while analyzing the example in Figure 2.

composition $\rho_{\tau_1} \circ \rho_{\tau_2}$ equals

$$\begin{aligned}\rho_{\tau_1} \circ \rho_{\tau_2} &= \exists x''.\ x \geq 0 \land x'' = x + 1 \land y'' = 1 \land \\ &\qquad\qquad y'' > x'' \land x' = x'' - 2 \land y' = y'' \\ &= x \geq 0 \land 1 > x + 1 \land x' = x - 1 \land y' = 1 \\ &= x \geq 0 \land 1 > x + 1 \\ &= x \geq 0 \land 0 > x \\ &= \mathsf{false}\ .\end{aligned}$$

Since false is well-founded, the counterexample $\pi$ is spurious because the candidate set $\mathcal{R}$ is too strong. The ranking relation that provides the evidence of $\rho_\pi$'s well-foundedness is the empty relation. Hence, we go to line 16, and add the empty relation $R_1 \equiv \emptyset$ to $\mathcal{R}$.

**Step III/Lines 7, 8, 10, 11, and 12:** We observe that the cyclic path $\pi = \tau_1\tau_2$ is still a spurious counterexample, since

$$\begin{aligned}\alpha_{\mathcal{P}}(\pi) &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \alpha_{\mathcal{P}}(\tau_2)) \\ &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \alpha_{\mathcal{P}}(y > x \land x' = x - 2 \land y' = y)) \\ &= \alpha_{\mathcal{P}}(\rho_{\tau_1} \circ \mathsf{true}) \\ &= \alpha_{\mathcal{P}}(\mathsf{true}) \\ &= \mathsf{true}\ ,\end{aligned}$$

and because true does not entail $R_1$. We go to line 8. Recall that $\rho_{\tau_1} \circ \rho_{\tau_2} =$ false. Because false $\subseteq R_1$, we detect that the counterexample $\pi$ is spurious due to imprecise abstraction. Hence, we go to line 10, and we collect the sets of predicates $\mathsf{Preds}(\rho_{\tau_2})$ and $\mathsf{Preds}(\rho_{\tau_1} \circ \rho_{\tau_2})$, see Table 1. The later set is empty, since $\rho_{\tau_1} \circ \rho_{\tau_2} =$ false. The set of predicates collected at line 11 is empty because $R_1$ is empty. Therefore, we finish this step with the following set of transition predicates:

$$\mathcal{P} = \{y > x, x' = x - 2, y' = y\}\ .$$

**Step IV/Lines 7, 8, 14, and 16:** We note that $\tau_1\tau_2$ is no longer a counterexample, because $\alpha_{\mathcal{P}}(\tau_1\tau_2) \subseteq R_1$. We find that for the cyclic path $\pi = \tau_3$ the abstract relation $\alpha_{\mathcal{P}}(\pi)$ does not entail any relations in $\mathcal{R}$. This means that $\pi$ is a counterexample. We do not know yet whether it is spurious. We therefore move to line 8. There does not exist a relation $R$ in $\mathcal{R}$ such that $\rho_\pi \subseteq R$. We therefore move to line 14. Recall that $\rho_{\tau_3} \equiv y \leq x \land x' = x \land y' = y + 1$. Using the techniques described in [20], we prove that $\rho_{\tau_3}$ is well-founded. We also compute a witness of $\rho_{\tau_3}$'s well-foundedness. The witness is a ranking relation $R_2$ such that $\rho_{\tau_3} \subseteq R_2$. We have

$$R_2 \equiv x - y \geq 0 \land x' - y' \leq x - y - 1\ .$$

Hence, $\pi$ is a spurious counterexample. We weaken $\mathcal{R}$ by adding $R_2$, at line 16.

**Step V/Lines 7, 8, 10, 11, and 12:** For $\pi = \tau_3$ we have $\alpha_{\mathcal{P}}(\pi) = \mathsf{true}$. Therefore $\alpha_{\mathcal{P}}(\pi)$ does not entail $R_2$. We know that $\rho_\pi \subseteq R_2$ (see Step IV). This means that $\tau_3$ is still a (spurious) counterexample wrt. the current abstraction. We refine the abstraction. The condition at line 8 succeeds, and we move to line 10. We collect the predicates from $\mathsf{Preds}(\rho_{\tau_3})$. At line 11, we collect the predicates from $\mathsf{Preds}(R_2)$, and $\mathsf{Preds}(\rho_{\tau_3} \circ R_2)$. After executing line 11, we have

$$\mathcal{P} = \{y > x, x' = x - 2, y' = y, y \leq x, x' = x, y' = y + 1,\\ x' - y' \leq x - y - 1, y \leq x - 1, x' - y' \leq x - y - 2\} \ .$$

**Step VI/Lines 7, 8, 14, and 16:** We observe that $\tau_3$ is no longer a counterexample, since $\alpha_{\mathcal{P}}(\tau_3) \subseteq R_2$. We consider the abstraction of the cyclic path $\pi = \tau_2\tau_1$. We have that $\alpha_{\mathcal{P}}(\pi)$ does not entail neither $R_1$ nor $R_2$. The relation $\rho_\pi$ such that

$$\rho_\pi \ = \ y > x \wedge x \geq 2 \wedge x' = x - 1 \wedge y' = 1$$

is well-founded, but is not contained in any $R \in \mathcal{R}$. Hence, $\pi$ is a spurious counterexample. Therefore we execute lines 14, and 16 of the algorithm, which weaken $\mathcal{R}$. A witness to the well-foundedness of $\rho_\pi$ is a ranking relation $R_3$ such that

$$R_3 \ \equiv \ x \geq 2 \wedge x' \leq x - 1 \ .$$

After executing line 16, we have $\mathcal{R} = \{R_1, R_2, R_3\}$.

**Step VII/Lines 7, 8, 10, 11, and 12:** Although $\rho_{\tau_2} \circ \rho_{\tau_1} \subseteq R_3$ we have $\alpha_{\mathcal{P}}(\tau_2 \circ \tau_1) \nsubseteq R_3$. This means that the abstraction is too coarse. Therefore, we execute lines 10, 11, and 12. At line 10, we collect the sets of predicates $\mathsf{Preds}(\rho_{\tau_1})$ and $\mathsf{Preds}(\rho_{\tau_2} \circ \rho_{\tau_1})$. At line 11, we collect the sets $\mathsf{Preds}(R_3)$, $\mathsf{Preds}(\rho_{\tau_1} \circ R_3)$, and $\mathsf{Preds}(\rho_{\tau_2} \circ \rho_{\tau_1} \circ R_3)$.

**Step VIII/Lines 7, 8, 14, and 16:** We observe that $\tau_2\tau_1$ is no longer a (spurious) counterexample. We discover that the relation $\rho_\pi$ corresponding to the cyclic path $\pi = \tau_1\tau_3\tau_2$ is well-founded, but is not contained in any relation $R \in \mathcal{R}$:

$$\rho_{\tau_1} \circ \rho_{\tau_3} \circ \rho_{\tau_2} \ = \ x = 0 \wedge x' = x - 1 \wedge y' = 2 \ .$$

This means that we found another spurious counterexample. Therefore we execute lines 8, 14 and 16. The ranking relation $R_4$ such that

$$R_4 \ \equiv \ x \geq 0 \wedge x' \leq x - 1$$

is a witness to the well-foundedness of $\rho_{\tau_1} \circ \rho_{\tau_3} \circ \rho_{\tau_2}$. After executing line 14, we have $\mathcal{R} = \{R_1, R_2, R_3, R_4\}$.

**Final Result:** For the abstraction $\alpha_{\mathcal{P}}(\pi)$ of every cyclic path $\pi$ there exists a relation $R$ in $\mathcal{R} = \{R_1, R_2, R_3, R_4\}$ such that $\alpha_{\mathcal{P}}(\pi)$ entails $R$. Therefore, the

algorithm terminates with $\mathcal{R} = \{R_1, R_2, R_3, R_4\}$ and the set of predicates $\mathcal{P}$ where

$$\begin{aligned} R_1 &= \mathsf{false} \ , \\ R_2 &= x - y \geq 0 \wedge x' - y' \leq x - y - 1 \ , \\ R_3 &= x \geq 2 \wedge x' \leq x - 1 \ , \\ R_4 &= x \geq 0 \wedge x' \leq x - 1 \ , \end{aligned}$$

and

$$\begin{aligned} \mathcal{P} = \{ & x \geq 0, x \geq 1, x \geq 2, x \geq 3, \\ & y \leq x, y \leq x - 1, y > x, \\ & x' = x + 1, x' = x, x' = x - 1, x' = x - 2, \\ & x' \leq x, x' \leq x - 1, x' \leq x - 3 \\ & x' - y' \leq x - y - 1, x' - y' \leq x - y - 2, \\ & y' = y + 1, y' = y, y' = 1 \} \ . \end{aligned}$$

## 6 Conclusion

Counterexample-guided abstraction refinement allows us to automatically extract just the information that is needed to prove the property. The crux of our abstraction refinement procedure for termination is the notion of a counterexample, and the different possible root causes when counterexamples are spurious.

We presented the first known counterexample-guided abstraction refinement algorithm for the proof of termination. We exploit recent results on transition invariants and transition predicate abstraction. Our counterexample-guided abstraction refinement algorithm successively weakens candidate transition invariants and successively refines abstractions.

**Future work** We are working on an implementation of this algorithm in Slam. Possible extensions of the algorithm presented here concern a wider class of properties (liveness with fairness assumptions) and a wider class of programs (concurrent and recursive programs); here the techniques described in [22] and in [23] can be useful.

**Acknowledgment** We thank Tom Ball, Aaron Bradley, and Lenore Zuck for discussions.

## References

1. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, pages 164–180. Springer, 2005.

2. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *IFM'2004: Fourth International Conference on Integrated Formal Methods*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004.
3. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *FMICS'02: Formal Methods for Industrial Critical Systems*, volume 66(2) of *ENTCS*, 2002.
4. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*. Springer, 2005.
5. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE'2003: Int. Conf. on Software Engineering*, pages 385–395, 2003.
6. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, December 1999.
7. M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS'2001: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 67–81. Springer, 2001.
8. M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV'2002: Computer Aided Verification*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.
9. P. Cousot. Partial completeness of abstract fixpoint checking. In *SARA'2000: Abstraction, Reformulation, and Approximation*, volume 1864 of *LNCS*, pages 1–15. Springer, 2000.
10. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, pages 1–24. Springer, 2005.
11. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'1977: Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'1978: Principles of Programming Languages*, pages 84–97. ACM Press, 1978.
13. S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *LICS'2001: Logic in Computer Science*, pages 51–60. IEEE, 2001.
14. J. Hatcliff and M. B. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *CONCUR'2001: Concurrency Theory*, volume 2154 of *LNCS*, pages 39–58. Springer, 2001.
15. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL'2004: Principles of Programming Languages*, pages 232–244. ACM Press, 2004.
16. F. Ivancic, H. Jain, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS'2005: Tools and Algorithms for Construction and Analysis of Systems*, LNCS. Springer, 2005. To appear.
17. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS'2001: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 98–112. Springer, 2001.
18. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.

19. K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV'2000: Computer Aided Verification*, volume 1855 of *LNCS*, pages 139–153. Springer, 2000.
20. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'2004: Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.
21. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS'2004: Logic in Computer Science*, pages 32–41. IEEE, 2004.
22. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL'2005: Principles of Programming Languages*, pages 132–144. ACM Press, 2005.
23. A. Podelski, I. Schaefer, and S. Wagner. Summaries for while programs with recursion. In S. Sagiv, editor, *ESOP'2005: European Symposium on Programming*, volume 3444 of *LNCS*, pages 94–107. Springer, 2005.
24. A. Tiwari. Termination of linear programs. In *CAV'2004: Computer Aided Verification*, volume 3114 of *LNCS*, pages 70–82. Springer, 2004.