Chapter 1
Predicate Abstraction for Program Verification

Ranjit Jhala[1], Andreas Podelski[2], and Andrey Rybalchenko[3,4]

Abstract We present basic principles of algorithms for the verification of safety and termination of programs. The algorithms call procedures on logical formulas in order to construct an abstraction and to refine an abstraction. The two underlying concepts are predicate abstraction and counterexample-guided abstraction refinement.

UC San Diego · University of Freiburg · Microsoft Research Cambridge · Technische Universität München

Contents

1 Predicate Abstraction for Program Verification 1
Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko
1.1 Introduction . . . 4
1.2 Definitions . . . 4
1.2.1 Programs . . . 5
1.2.2 Correctness: safety and termination . . . 8
1.3 Characterizing correctness via reachability . . . 8
1.3.1 Safety and reachability . . . 9
1.3.2 Termination and binary reachability . . . 9
1.4 Characterizing correctness via inductiveness . . . 11
1.4.1 Safety and closure under *post* . . . 11
1.4.2 Termination and transitive closure . . . 14
1.5 Abstraction . . . 16
1.5.1 Safety and predicate abstraction . . . 17
1.5.2 Termination and transition predicate abstraction . . . 22
1.6 Abstraction refinement . . . 28
1.6.1 Refinement of predicate abstraction . . . 29
1.6.2 Refinement of transition predicate abstraction . . . 35
1.7 Solving refinement constraints for predicate abstraction . . . 40
1.7.1 Least solution . . . 40
1.7.2 Greatest solution . . . 41
1.7.3 Intermediate solution using interpolation . . . 43
1.8 Tools . . . 44
1.9 Conclusion and discussion . . . 45
References . . . 45

1.1 Introduction

In this chapter, we are interested in program verification algorithms, i.e., in algorithms that take a program and a correctness property and try to answer the question whether the program is correct. Correctness is expressed by one of two properties of program executions: *safety* (which we formalize as the non-reachability of given *error* states), and termination. We are interested in a general class of programs for which safety and termination are not decidable. As a consequence, the algorithms must be based on abstraction.

The distinguishing feature of the algorithms is a specific way to call procedures over logical formulas in order to effectively construct an abstraction and to effectively refine an abstraction. The two underlying concepts are predicate abstraction and counterexample-guided abstraction refinement.

An abstraction maps a set of states to a superset. The terminology *predicate abstraction* refers to the fact that the superset is constructed from a basis of so-called *predicates* (pre-selected formulas that define sets of states). Now, with more predicates one has a larger choice for the construction of the superset, and the abstraction can be more precise. In this sense, adding more predicates refines the abstraction. The terminology *abstraction refinement* refers to the process of adding new predicates. The crux of the verification algorithms is the *counterexample-guided* procedure to select new predicates.

In the analogous way, we use *transition predicates* in order to construct the abstraction of a transition relation (a set of pairs of states).

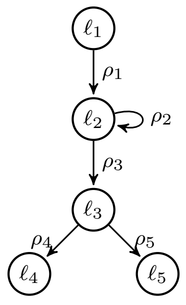
Program verification with predicate abstraction is an ongoing research topic. We can expect a great number of variations and optimizations to be proposed in the future. Yet, a few basic principles have emerged which will remain the basis for further developments even in the long term. Those few basic principles keep reappearing in different settings, each setting being motivated by a specific application scenario. The idea of this chapter is to abstract away from specific application scenarios and to present the few basic principles in the shortest possible way in the simplest possible formalism. For an exposition of the wealth of existing work in this area we refer to the survey in [42]. An account of the history of counterexample-guided abstraction refinement is given in [18].

1.2 Definitions

In this section, we use a formal setting based on logical formulas in order to introduce programs, computations, and two representative properties of computations, namely safety and termination.

```
main(int x, int y, int z) {
  assume(y >= z);
  while (x < y) {
    x++;
  }
  assert(x >= z);
}
```

(a)



(b)

$$\rho_1 = (goto(\ell_1, \ell_2) \wedge y \geq z \wedge unchanged(x, y, z))$$
$$\rho_2 = (goto(\ell_2, \ell_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge unchanged(y, z))$$
$$\rho_3 = (goto(\ell_2, \ell_3) \wedge x \geq y \wedge unchanged(x, y, z))$$
$$\rho_4 = (goto(\ell_3, \ell_4) \wedge x \geq z \wedge unchanged(x, y, z))$$
$$\rho_5 = (goto(\ell_3, \ell_5) \wedge x + 1 \leq z \wedge unchanged(x, y, z))$$

(c)

Fig. 1.1 An example program (a), its control flow graph (b), and its transition relations (c). Formally, the program is $\mathcal{P} = (V, pc, \varphi_{init}, \mathcal{T}, \varphi_{err})$ where $V = (pc, x, y, z)$ is the tuple of program variables, pc is the program counter variable, $\mathcal{T} = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$ is the set of transition relations, $\varphi_{init} = at\_\ell_1$ is the initial condition, and $\varphi_{err} = at\_\ell_5$ is the error condition. The primed variables are $V' = (pc', x', y', z')$. We use *goto* and *unchanged* as abbreviations. For example $goto(\ell_1, \ell_2)$ stands for $(pc = \ell_1 \wedge pc' = \ell_2)$ and $unchanged(x, y, z)$ stands for $(x' = x \wedge y' = y \wedge z' = z)$.

1.2.1 Programs

We specify a program formally through logical formulas. For an example, see Figure 1.1.

We assume a set V of logical variables that we call *program variables*. Each program variable comes with a *domain* (a set of values, e.g., integers).

The program counter pc is a distinguished program variable of every program, i.e., $pc \in V$. The domain of the program counter is a (finite) set Loc of special values called the *control locations* of the program.

A *program state s* is a function that assigns each program variable a value from its respective domain. Let Σ be the set of program states.

We sometimes fix an order on the variables by writing V as a tuple of variables, say $V = (pc, x, y, z)$, and then use a tuple of values to denote a state, e.g., $s = (\ell_1, 1, 3, 2)$.

A formula φ with free variables in V represents a set of program states. For example, the formula $pc = \ell$ represents the set of all states at the control location ℓ. The formula $x > 0$ represents the set of states (at any program location) where the program variable x has a value strictly greater than 0.

Each program variable (including *pc*) comes with its primed version. That is, for each program variable x in V, we have another variable x'. We write V' for the tuple of primed versions of program variables. A formula ψ with free variables in V and V' represents a set of pairs of states, i.e., a binary relation over states.

Formally, the pair (s_1, s_2) defines a valuation ν of variables in $V \cup V'$ where $\nu(x) = s_1(x)$ and $\nu(x') = s_2(x)$ for each variable x in V (and thus x' in V'). A formula ψ in unprimed and primed variables represents the set of pairs of states (s_1, s_2) such that the corresponding valuation ν satisfies ψ.

For example, the formula $pc = \ell_1 \wedge pc' = \ell_2$ represents the set of pairs of states (s_1, s_2) whose first component s_1 is a state at the control location ℓ_1 and whose second component s_2 is a state at the control location ℓ_2. The formula $x' = x$ represents the set of pairs of states (s_1, s_2) (at any program location) where the program variable x has the same value in the state s_1 and in the state s_2. The formula $x > 0 \wedge x' > x$ represents the set of pairs of states (s_1, s_2) where the program variable x has a value greater than 0 in the state s_1 and its value in the state s_1 is smaller than in the state s_2.

The formula $x' > 0$ represents the set of pairs of states (s_1, s_2) where the program variable x has a value greater than 0 in the state s_2 and its value in the state s_1 is unconstrained. Symmetrically, the formula $x > 0$ represents the set of pairs of states (s_1, s_2) where the program variable x has a value greater than 0 in the state s_1 and its value in the state s_2 is unconstrained.

We can use a formula φ in unprimed variables to represent both, a set of states and a binary relation over states. Thus, we can represent the restriction of the binary relation ψ to the set φ by the conjunction $\psi \wedge \varphi$.

To simplify the notation for transition relations, we introduce the following abbreviations (here ℓ, ℓ_1, and ℓ_2 are control locations and $x_1, \ldots, x_n$ are program variables).

$$\begin{aligned} \mathit{at\_}\ell &= (pc = \ell) \\ \mathit{at'\_}\ell &= (pc' = \ell) \\ \mathit{goto}(\ell_1, \ell_2) &= (\mathit{at\_}\ell_1 \wedge \mathit{at'\_}\ell_2) \\ \mathit{unchanged}(x_1, \ldots, x_n) &= (x_1' = x_1 \wedge \ldots \wedge x_n' = x_n) \end{aligned} \tag{1.1}$$

A program $\mathcal{P}$ is specified by the tuple $\mathcal{P} = (V, pc, \varphi_{init}, \mathcal{T}, \varphi_{err})$ consisting of the set of program variables V, the program counter *pc*, the initiation condition φ_{init}, the set of transition relations $\mathcal{T} = \{\rho_1, \ldots, \rho_n\}$, and the error condition φ_{err}.

The *initiation condition* φ_{init} and the *error condition* φ_{err} are formulas over variables in V. They represent the set of *initial states* and the set of *error states*, respectively.

The elements $\rho_1, \ldots, \rho_n$ are formulas over the program variables in V and their primed versions V'. If the formula ρ_i contains a conjunct of the form

$\mathit{goto}(\ell_1, \ell_2)$ for two locations ℓ_1 and ℓ_2, we say that ρ_i is a *transition* from ℓ_1 to ℓ_2.

The set of transition relations $\mathcal{T} = \{\rho_1, \dots, \rho_n\}$ defines the *program transition relation* of $\mathcal{P}$, which is represented by the formula

$$\rho_{\mathcal{P}} = \rho_1 \vee \dots \vee \rho_n \ . \tag{1.2}$$

The formula $\rho_{\mathcal{P}}$ thus represents the union of the transition relations represented by the transitions $\rho_1, \dots, \rho_n$.

Example 1. Our example program has an initiation condition $\varphi_{init} = (at\_\ell_1)$ and an error condition $\varphi_{err} = (at\_\ell_5)$. That is, every state at control location ℓ_1 is an initial state and every state at control location ℓ_5 is an error state. The set of program transitions $\mathcal{T} = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$ corresponds to the graph as shown in Figure 1.1(b). We call this graph the *control flow graph* of the program. The transition relations $\rho_1, \rho_2, \rho_3, \rho_4$, and ρ_5 are defined in Figure 1.1(c). The transition relation of the program is the disjunction $\rho_{\mathcal{P}} = \rho_1 \vee \rho_2 \vee \rho_3 \vee \rho_4 \vee \rho_5$. $\square$

It is convenient to identify formulas with the sets and relations that they represent. Accordingly, we identify the logical consequence relation (entailment) between formulas $\models$ with the set inclusion $\subseteq$ between the sets that they represent. (All examples presented in this chapter use the theory of linear rational arithmetic.) Furthermore, we identify the satisfaction relation between a valuation and a formula (which is also denoted by $\models$) with the membership relation $\in$ between the corresponding state and the represented set of states (or between the corresponding pair of states and the represented relation between states).

Often, a formal setting for program verification is based on the notion of a *control flow graph*, i.e., a graph whose nodes correspond to the program locations and whose edges are labeled by statements. This may reflect a particular design decision in a practical implementation. It is clear, however, that one can derive the logical formula denoting the transition relation of the program from a control flow graph, and vice versa. As in the example, the logical formula denoting the transition relation of the program induces a graph where each edge (ℓ_1, ℓ_2) arises from a conjunct $\mathit{goto}(\ell_1, \ell_2)$ in the logical formula. By starting directly with logical formulas, we obtain a uniform setting.

Example 2. For example, we consider the program shown in Figure 1.1. Let s be a program state given by the tuple $(\ell_1, 1, 3, 2)$ (which stands for the mapping that assigns 1, 3, 2, and ℓ_1 to the program variables x, y, z, and pc, respectively). Then, we have $s \models y \geq z$ (or, written differently, $s \in y \geq z$). Furthermore, we have $y \geq z \models y + 1 \geq z$ (or, written differently, $y \geq z \subseteq y + 1 \geq z$). $\square$

1.2.2 Correctness: safety and termination

Given a progam $\mathcal{P}$ with the program transition relation $\rho_{\mathcal{P}}$, the set of initial states φ_{init}, and the set of error states φ_{err}, we formalize program correctness as a property of program computations. A *program computation* of $\mathcal{P}$ is either a finite sequence $s_1, \ldots, s_n$ or an infinite sequence $s_1, s_2, \ldots$ of states that is generated by the program transition relation $\rho_{\mathcal{P}}$, starts in an initial state, and if it is finite then it can not be continued after the last state (s_n is a *deadlock* state). This means:

- each pair of consecutive states s_i and s_{i+1} in the sequence is an element of the program transition relation, i.e., $(s_i, s_{i+1}) \in \rho_{\mathcal{P}}$,
- the first element of the sequence is an initial state, i.e., $s_1 \in \varphi_{init}$,
- if the sequence is finite with s_n as its last element, then the state s_n does not have any successor state wrt. the program transition relation $\rho_{\mathcal{P}}$, i.e., there is no state s such that $(s_n, s) \in \rho_{\mathcal{P}}$.

Example 3. The (finite) sequence of states below is a program computation in our example program $\mathcal{P}$.

$$(\ell_1, 1, 3, 2), (\ell_2, 1, 3, 2), (\ell_2, 2, 3, 2), (\ell_2, 3, 3, 2), (\ell_3, 3, 3, 2), (\ell_4, 3, 3, 2)$$

The sequence of states starts in an initial state and follows the sequence of transitions $\rho_1, \rho_2, \rho_2, \rho_3, \rho_4$. The last state in the sequence does not have any successor state wrt. the program transition relation $\rho_{\mathcal{P}}$. □

The verification of a large class of properties of program computations can be reduced to reasoning about safety and to reasoning about termination.

A program is *safe* if no error state occurs in any program computation. A program *terminates* if every program computation is finite.

In general, the length of program computations is unbounded even if the program is terminating (see, for example, the program in Figure 1.1). Thus, we will not try to verify termination by checking the existence of a bound on the computation length. Let us note, in passing, that a method based on a reduction to finite-state model checking amounts to checking boundedness, since a finite-state program terminates if and only if the length of program computations is bounded.

1.3 Characterizing correctness via reachability

We will next characterize safety and termination by conditions that are suitable for the abstraction-based verification of safety and termination. The conditions are defined in terms of reachability of states and, respectively, reachability of pairs of states (binary reachability).

1.3.1 *Safety and reachability*

A state s is *reachable* if there exists a program computation $s_1, s_2, \ldots$ with an occurence of s (i.e., there exists a position i such that $s_i = s$). We use

$$\varphi_{reach}$$

for the set of all reachable states.

An *invariant* is a set φ that contains all reachable states, i.e., $\varphi_{reach} \subseteq \varphi$.

The program $\mathcal{P}$ is *safe* if and only if the complement of the set of error states is an invariant, i.e., if

$$\varphi_{reach} \subseteq \Sigma \setminus \varphi_{err} \ . \tag{1.3}$$

Example 4. For our example program, the set of reachable states is shown below.

$$\begin{aligned}\varphi_{reach} = (&at\_\ell_1 \lor (at\_\ell_2 \land y \geq z) \lor \\ &(at\_\ell_3 \land y \geq z \land x \geq y) \lor (at\_\ell_4 \land y \geq z \land x \geq y))\end{aligned}$$

This set does not contain any error states, i.e., we have

$$\varphi_{reach} \subseteq \neg at\_\ell_5 \ .$$

□

In Section 1.4.1, we show that one can construct the set φ_{reach} by an iterative application of a function on sets of states. In general, one needs to iterate the application of the function infinitely many times. In Section 1.5.1, we show that one can construct a superset of φ_{reach} by an iterative application of an abstraction of the function. We construct the abstract function automatically using *predicate abstraction*. With predicate abstraction, one needs to iterate the application of the abstract function only finitely many times.

1.3.2 *Termination and binary reachability*

We extend the notion of reachability from states to pairs of states. A pair of states (s, s') is *reachable* if s is reachable from the initial state and s' is reachable from s. Which is, if there exists a program computation in which s is followed by s' (i.e., the program computation is of the form $s_1, \ldots, s_i, \ldots, s_j, \ldots$ where $s_i = s$ and $s_j = s'$ for positions i and j such that $1 \leq i < j$). We use

$$\psi_{reach}$$

for the set of reachable pairs of states and call it the *binary reachability* relation.

A *transition invariant* is a binary relation over states ψ that contains the binary reachability relation, i.e., $\psi_{reach} \subseteq \psi$.

Just as we used the notion of invariant to characterize safety, we will use the notion of a transition invariant to characterize termination. The interest of the characterization of termination in this way lies in a proof method for termination which parallels the proof method for safety. As we show in Section 1.4.2, one can construct the set ψ_{reach} by an iterative application of a function on sets of pairs of states. In general, one needs to iterate the application of the function infinitely many times. In Section 1.5.2, we show that one can construct a superset of ψ_{reach} by an iterative application of an abstraction of the function. We construct the abstract function automatically using the analogue of predicate abstraction for *transition predicate*s. One needs to iterate the application of the abstract function only finitely many times.

The termination of a program can be equivalently expressed as the well-foundedness of its program transition relation. A binary relation ψ is defined to be *well-founded* if it does not generate any infinite sequence (i.e., if there is no infinite sequence $s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in \psi$ for all $i = 1, 2, ...$). For example, the relation $x > 0 \wedge x' > x$ is well-founded. The union of well-founded relations is in general not well-founded (take, for example, the union of the relations $x > 0 \wedge x' > x$ and $y > 0 \wedge y' > y$).

Assume we are given a number of well-founded relations $\psi_1, \ldots, \psi_n$ (each corresponding, for example, to the program transition relation of a terminating program). The program $\mathcal{P}$ is terminating if the union of the n well-founded relations, which we call a disjunctively well-founded relation, is a transition invariant, i.e., if

$$\psi_{reach} \subseteq \psi_1 \cup \ldots \cup \psi_n \ . \tag{1.4}$$

The proof of this fact relies in Ramsey's theorem on combinatorics for infinite graphs, see [52].

Just as we used the notion of invariant to characterize safety, we have used the notion of a transition invariant to characterize termination. The interest of the characterization of termination by transition invariants lies in a proof method for termination which parallels the proof method for safety. As we show in Section 1.4.2, one can construct the set ψ_{reach} by an iterative application of a function on sets of pairs of states. In general, one needs to iterate the application of the function infinitely many times. In Section 1.5.2, we show that one can construct a superset of ψ_{reach} by an iterative application of an abstraction of the function. We construct the abstract function automatically using the analogue of predicate abstraction for *transition predicate*s. One needs to iterate the application of the abstract function only finitely many times.

To be precise, we have characterized termination by the fact that the union of a (finite) number of well-founded relations forms a transition invariant. We have not said where the well-founded relations come from. For the purpose of this presentation, we assume that they are given. There are, however, many strategies to obtain formulas that represent the required well-founded relations; see, e.g., [19, 52].

1.4 Characterizing correctness via inductiveness

In order to check reachability (or binary reachability), we need to construct the set of reachable states (or the set of reachable pairs of states). The construction is possible, in theory, by the iterative application of a function over sets of states (or a function over sets of pairs of states). This construction may need infinitely many iterations. It defines the smallest set that is *inductive*, i.e., closed under the application of the function. We may not need to construct the smallest set. It may be sufficient to construct a superset. The only way to show that a given set is indeed a superset, i.e., that it contains the set of reachable states (or the set of reachable pairs of states), is to show that it is inductive.

1.4.1 Safety and closure under post

Let φ be a formula over V and let ρ be a formula over V and V'. We define a *post-condition* function *post* as follows.

$$\mathit{post}(\varphi, \rho) \ = \ \exists V'' : \varphi[V''/V] \land \rho[V''/V][V/V'] \tag{1.5}$$

Here $\varphi[V''/V]$ represents the result of replacing V by V'' in φ, while $\rho[V''/V][V/V']$ requires first replacing V by V'' and then replacing V' by V. An application $\mathit{post}(\varphi, \rho)$ computes the image of the set φ under the relation ρ. We observe the following useful property of the post-condition function.

$$\begin{aligned} &\forall \varphi\, \forall \rho_1\, \forall \rho_2 : \mathit{post}(\varphi, \rho_1 \lor \rho_2) = (\mathit{post}(\varphi, \rho_1) \lor \mathit{post}(\varphi, \rho_2)) \\ &\forall \varphi_1\, \forall \varphi_2\, \forall \rho : \mathit{post}(\varphi_1 \lor \varphi_2, \rho) = (\mathit{post}(\varphi_1, \rho) \lor \mathit{post}(\varphi_2, \rho)) \end{aligned} \tag{1.6}$$

This property states that the post-condition computation distributes over disjunction wrt. each argument.

Furthermore, for a natural number n we define $\mathit{post}^n(\varphi, \rho)$ to represent the n-fold application of the *post* function to φ with respect to ρ. Formally, we have:

$$post^n(\varphi, \rho) \;=\; \begin{cases} \varphi & \text{if } n = 0 \\ post(post^{n-1}(\varphi, \rho), \rho) & \text{otherwise} \end{cases} \tag{1.7}$$

Example 5. For example, given the transition relation ρ_2 and the program variables $V = (pc, x, y, z)$ from our example program, we compute the following post condition.

$$\begin{aligned}
&post(at\_\ell_2 \wedge y \geq z, \rho_2) \\
&\quad = (\exists V'' : (at\_\ell_2 \wedge y \geq z)[V''/V] \wedge \rho_2[V''/V][V/V']) \\
&\quad = (\exists V'' : (pc'' = \ell_2 \wedge y'' \geq z'') \wedge \\
&\qquad\qquad (pc'' = \ell_2 \wedge pc' = \ell_2 \wedge x'' + 1 \leq y'' \wedge x' = x'' + 1 \wedge \\
&\qquad\qquad\; y' = y'' \wedge z' = z'')[V/V']) \\
&\quad = (\exists V'' : (pc'' = \ell_2 \wedge y'' \geq z'') \wedge \\
&\qquad\qquad (pc'' = \ell_2 \wedge pc = \ell_2 \wedge x'' + 1 \leq y'' \wedge x = x'' + 1 \wedge \\
&\qquad\qquad\; y = y'' \wedge z = z'')) \\
&\quad = (pc = \ell_2 \wedge y \geq z \wedge x \leq y)
\end{aligned}$$

We compute the 2-fold application by reusing the above result.

$$\begin{aligned}
&post^2(at\_\ell_2 \wedge y \geq z, \rho_2) \\
&\quad = post(post(at\_\ell_2 \wedge y \geq z, \rho_2), \rho_2) \\
&\quad = post(pc = \ell_2 \wedge y \geq z \wedge x \leq y, \rho_2) \\
&\quad = (\exists V'' : (pc'' = \ell_2 \wedge y'' \geq z'' \wedge x'' \leq y'') \wedge \\
&\qquad\qquad (pc'' = \ell_2 \wedge pc = \ell_2 \wedge x'' + 1 \leq y'' \wedge x = x'' + 1 \wedge \\
&\qquad\qquad\; y = y'' \wedge z = z'')) \\
&\quad = (pc = \ell_2 \wedge y \geq z \wedge x - 1 \leq y \wedge x \leq y) \\
&\quad = (pc = \ell_2 \wedge y \geq z \wedge x \leq y)
\end{aligned}$$

□

We characterize φ_{reach} using *post* as follows.

$$\begin{aligned}
\varphi_{reach} &= \varphi_{init} \vee post(\varphi_{init}, \rho_{\mathcal{P}}) \vee post(post(\varphi_{init}, \rho_{\mathcal{P}}), \rho_{\mathcal{P}}) \vee \ldots \\
&= \textstyle\bigvee_{i \geq 0} post^i(\varphi_{init}, \rho_{\mathcal{P}})
\end{aligned} \tag{1.8}$$

The above disjunction (over every number of applications of the post-condition function) ensures that all reachable states are taken into consideration.

Example 6. We compute φ_{reach} for our example program. We first obtain the post-condition after applying the transition relation of the program once.

$$\begin{aligned}
&post(at\_\ell_1, \rho_{\mathcal{P}})\\
&\quad = (post(at\_\ell_1, \rho_1) \vee post(at\_\ell_1, \rho_2) \vee post(at\_\ell_1, \rho_3) \vee\\
&\qquad post(at\_\ell_1, \rho_4) \vee post(at\_\ell_1, \rho_5))\\
&\quad = post(at\_\ell_1, \rho_1)\\
&\quad = (at\_\ell_2 \wedge y \geq z)
\end{aligned}$$

Next, we obtain the post-condition for one more application.

$$\begin{aligned}
&post(at\_\ell_2 \wedge y \geq z, \rho_{\mathcal{P}})\\
&\quad = (post(at\_\ell_2 \wedge y \geq z, \rho_2) \vee post(at\_\ell_2 \wedge y \geq z, \rho_3))\\
&\quad = (at\_\ell_2 \wedge y \geq z \wedge x \leq y \vee at\_\ell_3 \wedge y \geq z \wedge x \geq y)
\end{aligned}$$

We repeat the application step once again.

$$\begin{aligned}
&post(at\_\ell_2 \wedge y \geq z \wedge x \leq y \vee at\_\ell_3 \wedge y \geq z \wedge x \geq y, \rho_{\mathcal{P}})\\
&\quad = (post(at\_\ell_2 \wedge y \geq z \wedge x \leq y, \rho_{\mathcal{P}}) \vee post(at\_\ell_3 \wedge y \geq z \wedge x \geq y, \rho_{\mathcal{P}}))\\
&\quad = (post(at\_\ell_2 \wedge y \geq z \wedge x \leq y, \rho_2) \vee post(at\_\ell_2 \wedge y \geq z \wedge x \leq y, \rho_3) \vee\\
&\qquad post(at\_\ell_3 \wedge y \geq z \wedge x \geq y, \rho_4) \vee post(at\_\ell_3 \wedge y \geq z \wedge x \geq y, \rho_5))\\
&\quad = (at\_\ell_2 \wedge y \geq z \wedge x \leq y \vee at\_\ell_3 \wedge y \geq z \wedge x = y \vee\\
&\qquad at\_\ell_4 \wedge y \geq z \wedge x \geq y)
\end{aligned}$$

So far, by iteratively applying the post-condition function to φ_{init} we obtained the following disjunction.

$$\begin{aligned}
&at\_\ell_1 \vee\\
&at\_\ell_2 \wedge y \geq z \vee\\
&at\_\ell_2 \wedge y \geq z \wedge x \leq y \vee at\_\ell_3 \wedge y \geq z \wedge x \geq y \vee\\
&at\_\ell_2 \wedge y \geq z \wedge x \leq y \vee at\_\ell_3 \wedge y \geq z \wedge x = y \vee\\
&at\_\ell_4 \wedge y \geq z \wedge x \geq y
\end{aligned}$$

We present this disjunction in a logically equivalent, simplified form as follows.

$$\begin{aligned}
&at\_\ell_1 \vee\\
&at\_\ell_2 \wedge y \geq z \vee\\
&at\_\ell_3 \wedge y \geq z \wedge x \geq y \vee\\
&at\_\ell_4 \wedge y \geq z \wedge x \geq y
\end{aligned}$$

Any further application of the post-condition function does not produce any additional disjuncts. Hence, φ_{reach} is the above disjunction. □

Inductive proof of safety

An *inductive invariant* φ contains the initial states and is closed under successors [30, 40]. Formally, an inductive invariant is a formula over the program variables that represents a superset of the initial program states and is closed under the application of the *post* function wrt. the relation $\rho_{\mathcal{P}}$, i.e.,

$$\varphi_{init} \models \varphi \quad \text{and} \quad \mathit{post}(\varphi, \rho_{\mathcal{P}}) \models \varphi \ .$$

A program is safe if there exists an inductive invariant φ that does not contain any error states, i.e., $\varphi \wedge \varphi_{err} \models \mathit{false}$.

Example 7. For our example program, the weakest inductive invariant consists of the set of all states and is represented by the formula *true*. The strongest inductive invariant was obtained in Example 6. The strongest inductive invariant does not contain any error states. We observe that a slightly weaker inductive invariant below also proves the safety of our examples.

$$\mathit{at\_}\ell_1 \vee (\mathit{at\_}\ell_2 \wedge y \geq z) \vee (\mathit{at\_}\ell_3 \wedge y \geq z \wedge x \geq y) \vee \mathit{at\_}\ell_4$$

□

1.4.2 Termination and transitive closure

Let ρ_1 and ρ_2 be formulas over V and V'. We define a *relational composition* function $\circ$ as follows.

$$\rho_1 \circ \rho_2 \ = \ \exists V'' : \rho_1[V''/V'] \wedge \rho_2[V''/V] \qquad (1.9)$$

Example 8. For example, given the transition relations ρ_1, ρ_2, and the program variables $V = (pc, x, y, z)$ from our example program we obtain the following relational composition.

$$\begin{aligned}
\rho_1 \circ \rho_2 = (\exists V'' : (&pc = \ell_1 \wedge pc' = \ell_2 \wedge y \geq z \wedge \\
&x' = x \wedge y' = y \wedge z' = z)[V''/V'] \wedge \\
&(pc = \ell_2 \wedge pc' = \ell_2 \wedge x + 1 \leq y \wedge \\
&\ x' = x + 1 \wedge y' = y \wedge z' = z)[V''/V]) \\
= (\exists V'' : (&pc = \ell_1 \wedge pc'' = \ell_2 \wedge y \geq z \wedge \\
&x'' = x \wedge y'' = y \wedge z'' = z) \wedge \\
&(pc'' = \ell_2 \wedge pc' = \ell_2 \wedge x'' + 1 \leq y'' \wedge \\
&x' = x'' + 1 \wedge y' = y'' \wedge z' = z'')) \\
= (pc = \ell_1 \wedge{}& pc' = \ell_2 \wedge y \geq z \wedge x + 1 \leq y \wedge \\
x' = x + 1 \wedge{}& y' = y \wedge z' = z)
\end{aligned}$$

□

For a given $\rho_{\mathcal{P}}$, a binary relation ψ and a natural number n, we define an *n-time transition composition* $comp^n(\rho_{\mathcal{P}})$ of $\rho_{\mathcal{P}}$ with ψ as follows.

$$comp^n(\psi) \;=\; \begin{cases} \psi & \text{if } n = 0 \\ comp^{n-1}(\psi) \circ \rho_{\mathcal{P}} & \text{otherwise} \end{cases}$$

We can compute the (irreflexive) transitive closure $\rho_{\mathcal{P}}^+$ using *comp* as follows.

$$\begin{aligned} \rho_{\mathcal{P}}^+ &= \rho_{\mathcal{P}} \vee \rho_{\mathcal{P}} \circ \rho_{\mathcal{P}} \vee \rho_{\mathcal{P}} \circ \rho_{\mathcal{P}} \circ \rho_{\mathcal{P}} \vee \ldots \qquad (1.10) \\ &= \textstyle\bigvee_{i \geq 1} comp^i(\rho_{\mathcal{P}}) \end{aligned}$$

We will be using a restriction of $\rho_{\mathcal{P}}^+$ to reachable states φ_{reach}. For this reason, we define

$$\psi_{ti} \;=\; \bigvee_{i \geq 1} comp^i(\varphi_{reach} \wedge V' = V) \qquad (1.11)$$

That is, ψ_{ti} is a transition invariant that is characterized using iteration of relational composition.

Inductive proof for termination

The restriction of the program transition relation $\rho_{\mathcal{P}}$ to the reachable program states is given by $\rho_{\mathcal{P}} \wedge \varphi_{reach}$ (the conjunction of a formula over V and V' and a formula over V). A program terminates if and only if the binary relation $\rho_{\mathcal{P}} \wedge \varphi_{reach}$ is well-founded.

Example 9. For our example, we obtain the following restriction of the program transition relation to reachable states.

$$\begin{aligned} \rho_{\mathcal{P}} \wedge \varphi_{reach} = (&goto(\ell_1, \ell_2) \wedge y \geq z \wedge unchanged(x, y, z) \vee \\ &goto(\ell_2, \ell_2) \wedge y \geq z \wedge x + 1 \leq y \wedge x' = x + 1 \wedge unchanged(y, z) \vee \\ &goto(\ell_2, \ell_3) \wedge y \geq z \wedge x \geq y \wedge unchanged(x, y, z) \vee \\ &goto(\ell_3, \ell_4) \wedge y \geq z \wedge x \geq y \wedge x \geq z \wedge unchanged(x, y, z)) \end{aligned}$$

The restriction consists of four disjuncts, since the transition relation ρ_5 does not intersect with φ_{reach}. Furthermore, the restriction is well-founded, i.e., our program terminates. Any attempt to construct an infinite sequence leads to unbounded increase of the values of the variable x, which contradicts the condition that x is bounded from above by y whenever the loop execution is carried on. □

An *inductive transition invariant* ψ contains the restriction of the program transition relation to reachable states and is closed under relational composition with the program transition relation [52]. Formally, given an inductive invariant φ, we require that an inductive transition invariant ψ satisfies the following conditions:

$$\varphi \wedge \rho_{\mathcal{P}} \models \psi \quad \text{and} \quad \psi \circ \rho_{\mathcal{P}} \models \psi \ .$$

A program terminates if there exist a finite number of well-founded relations $\psi_1, \ldots, \psi_n$ whose union contains an inductive transition invariant, i.e., $\psi \models \psi_1 \vee \ldots \vee \psi_n$.

1.5 Abstraction

The computation of the set of reachable program states requires the iterative application of the post-condition function on the initial program states, see Equation (1.8). The iteration stops when no new disjuncts are being added. Unfortunately, in many cases, the iteration will never stop.

Example 10. We consider the iterative computation of the set of states that is reachable from $at\_\ell_2 \wedge x \leq z$ by applying the transition ρ_2 of our example program. We obtain the following sequence of post-conditions (where $V = (pc, x, y, z)$).

$$\begin{aligned}
post(at\_\ell_2 \wedge x \leq z, \rho_2) &= (\exists V'' : (pc'' = \ell_2 \wedge x'' \leq z'') \wedge \\
&\qquad\qquad (pc'' = \ell_2 \wedge pc = \ell_2 \wedge x'' + 1 \leq y'' \wedge \\
&\qquad\qquad\ x = x'' + 1 \wedge y = y'' \wedge z = z'')) \\
&= (at\_\ell_2 \wedge x - 1 \leq z \wedge x \leq y) \\
post^2(at\_\ell_2 \wedge x \leq z, \rho_2) &= (at\_\ell_2 \wedge x - 2 \leq z \wedge x \leq y) \\
post^3(at\_\ell_2 \wedge x \leq z, \rho_2) &= (at\_\ell_2 \wedge x - 3 \leq z \wedge x \leq y) \\
&\ldots \\
post^n(at\_\ell_2 \wedge x \leq z, \rho_2) &= (at\_\ell_2 \wedge x - n \leq z \wedge x \leq y)
\end{aligned}$$

In this sequence, we observe that at each iteration yields a set of states that contains states not discovered before. For example, the set of states reachable after applying the post-condition function once is not included in the original set, i.e.,

$$(at\_\ell_2 \wedge x - 1 \leq z \wedge x \leq y) \not\models (at\_\ell_2 \wedge x \leq z) \ .$$

The set of states reachable after applying the post-condition function twice is not included in the union of the above two sets, i.e.,

$$(at\_\ell_2 \wedge x - 2 \leq z \wedge x \leq y) \not\models (at\_\ell_2 \wedge x - 1 \leq z \wedge x \leq y \vee at\_\ell_2 \wedge x \leq z) \ .$$

Furthermore, we observe that the set of states reachable after n-fold application of *post*, where $n \geq 1$, still contains previously unreached states, i.e.,

$$\begin{aligned}\forall n \geq 1 : (&at\_\ell_2 \wedge x - n \leq z \wedge x \leq y) \\ &\not\models (at\_\ell_2 \wedge x \leq z \vee \textstyle\bigvee_{1 \leq i < n}(at\_\ell_2 \wedge x - i \leq z \wedge x \leq y)) \ .\end{aligned}$$

□

A similar example can be used to show the possibility of non-termination for the procedure which constructs the strongest transition invariant.

1.5.1 Safety and predicate abstraction

Instead of computing φ_{reach}, we compute an over-approximation of φ_{reach} by a superset $\varphi^{\#}_{reach}$. Then, we check whether $\varphi^{\#}_{reach}$ contains any error states. If $\varphi^{\#}_{reach} \wedge \varphi_{err} \models \mathit{false}$ holds then $\varphi_{reach} \wedge \varphi_{err} \models \mathit{false}$. Hence the program is safe.

Similarly to the iterative computation of φ_{reach}, we compute $\varphi^{\#}_{reach}$ by applying iteration. However, instead of iteratively applying the post-condition function *post* we use its over-approximation $post^{\#}$ such that

$$\forall \varphi\ \forall \rho : post(\varphi, \rho) \models post^{\#}(\varphi, \rho) \ . \tag{1.12}$$

We decompose the computation of $post^{\#}$ into two steps. First, we apply *post* and then, we over-approximate the result using a function α such that

$$\forall \varphi : \varphi \models \alpha(\varphi) \ . \tag{1.13}$$

That is, given an over-approximating function α we define $post^{\#}$ as follows.

$$post^{\#}(\varphi, \rho) \ = \ \alpha(post(\varphi, \rho)) \tag{1.14}$$

Finally, we obtain $\varphi^{\#}_{reach}$:

$$\begin{aligned}\varphi^{\#}_{reach} &= \alpha(\varphi_{init}) \vee \\ &\quad post^{\#}(\alpha(\varphi_{init}), \rho_{\mathcal{P}}) \vee \\ &\quad post^{\#}(post^{\#}(\alpha(\varphi_{init}), \rho_{\mathcal{P}}), \rho_{\mathcal{P}}) \vee \dots \\ &= \textstyle\bigvee_{i \geq 0}(post^{\#})^i(\alpha(\varphi_{init}), \rho_{\mathcal{P}})\end{aligned} \tag{1.15}$$

We formalize our over-approximation based reachability computation as follows. The set of reachable program states is contained in the result of abstract post condition computation given by Equation (1.15). Formally, $\varphi_{reach} \models \varphi^{\#}_{reach}$.

Predicate abstraction We construct an over-approximation using a given set of building blocks, so-called predicates. Each predicate is a formula over the program variables V.

We fix a finite set of predicates $\mathit{Preds} = \{p_1, \dots, p_n\}$. Then, we define an over-approximation of φ that is constructed using *Preds* as follows [23, 31].

$$\alpha(\varphi) = \bigwedge\{p \in \mathit{Preds} \mid \varphi \models p\} \tag{1.16}$$

If the set of entailed predicates is empty then the result of applying predicate abstraction is $\bigwedge \emptyset$ and is equivalent to *true*.

Example 11. For example, we consider a set of predicates $\mathit{Preds} = \{at\_\ell_1, \dots, at\_\ell_5, y \geq z, x \geq y\}$. We compute $\alpha(at\_\ell_2 \wedge y \geq z \wedge x + 1 \leq y)$ as follows. First, we check the logical consequence between the argument to the abstraction function and each of the predicates. The results are presented in the following table.

	at_ℓ_1	at_ℓ_2	at_ℓ_3	at_ℓ_4	at_ℓ_5	$y \geq z$	$x \geq y$
$at\_\ell_2 \wedge y \geq z \wedge x + 1 \leq y$	$\not\models$	$\models$	$\not\models$	$\not\models$	$\not\models$	$\models$	$\not\models$

Then, we take the conjunction of the entailed predicates as the result of the abstraction.

$$\alpha(at\_\ell_2 \wedge y \geq z \wedge x + 1 \leq y) = \bigwedge\{at\_\ell_2, y \geq z\} = at\_\ell_2 \wedge y \geq z$$

□

The predicate abstraction function in Equation (1.16) approximates φ using a conjunction of predicates, which requires n entailment checks where n is the number of given predicates.

Example 12. We use predicate abstraction to compute $\varphi^{\#}_{reach}$ for our example program following the iterative scheme presented in Equation (1.15). Let $\mathit{Preds} = \{\mathit{false}, at\_\ell_1, \dots, at\_\ell_5, y \geq z, x \geq y\}$. First, let φ_1 be the over-approximation of the set of initial states φ_{init}:

$$\varphi_1 = \alpha(at\_\ell_1) = \bigwedge\{at\_\ell_1\} = at\_\ell_1\ .$$

We apply $post^{\#}$ on φ_1 wrt. each program transition and obtain

$$\varphi_2 = post^{\#}(\varphi_1, \rho_1) = \alpha(\underbrace{at\_\ell_2 \wedge y \geq z}_{post(\varphi_1, \rho_1)}) = \bigwedge\{at\_\ell_2, y \geq z\} = at\_\ell_2 \wedge y \geq z\ ,$$

whereas $post^{\#}(\varphi_1, \rho_2) = \dots = post^{\#}(\varphi_1, \rho_5) = \bigwedge\{\mathit{false}, \dots\} = \mathit{false}$.

Now we apply program transitions on φ_2 using $post^{\#}$. The application of ρ_1, ρ_4, and ρ_5 on φ_2 results in *false* for the following reason. φ_2 requires

at_ℓ_2, but the transition relations ρ_1, ρ_4, and ρ_5 are applicable if either at_ℓ_1 or at_ℓ_3 holds. For ρ_2 we obtain

$$post^\#(\varphi_2, \rho_2) = \alpha(at\_\ell_2 \wedge y \geq z \wedge x \leq y) = \bigwedge\{at\_\ell_2, y \geq z\} = at\_\ell_2 \wedge y \geq z \ .$$

The resulting set above is equal to φ_2 and, therefore, is discarded, since we are already exploring states reachable from φ_2. For ρ_3 we obtain

$$\begin{aligned} post^\#(\varphi_2, \rho_3) &= \alpha(at\_\ell_3 \wedge y \geq z \wedge x \geq y) \\ &= \bigwedge\{at\_\ell_3, y \geq z, x \geq y\} = at\_\ell_3 \wedge y \geq z \wedge x \geq y \\ &= \varphi_3 \ . \end{aligned}$$

We compute an over-approximation of the set of states that are reachable from φ_3 by applying $post^\#$. The transitions ρ_1, ρ_2, and ρ_3 results in *false* due to an inconsistency caused by the program counter valuations in φ_3 and the respective transition relations. For the transition ρ_4 we obtain

$$\begin{aligned} post^\#(\varphi_3, \rho_4) &= \alpha(at\_\ell_4 \wedge y \geq z \wedge x \geq y \wedge x \geq z) \\ &= \bigwedge\{at\_\ell_4, y \geq z, x \geq y\} = at\_\ell_4 \wedge y \geq z \wedge x \geq y \\ &= \varphi_4 \ . \end{aligned}$$

For the transition ρ_5, which corresponds to the assertion violation, we obtain

$$\begin{aligned} post^\#(\varphi_3, \rho_5) &= \alpha(at\_\ell_5 \wedge y \geq z \wedge x \geq y \wedge x + 1 \leq z) \\ &= \mathit{false} \ . \end{aligned}$$

Any further application of program transitions does not compute any additional reachable states. We conclude that $\varphi^\#_{reach} = \varphi_1 \vee \ldots \vee \varphi_4$. Furthermore, since $\varphi^\#_{reach} \wedge at\_\ell_5 \models \mathit{false}$ the program is safe. □

Algorithm AbstReach We combine the characterization of abstract reachability using Equation (1.15) with the predicate abstraction function given in Equation (1.16) and obtain an algorithm AbstReach for computing *ReachStates*$^\#$. The algorithm is shown in Figure 1.2.

AbstReach takes as input a finite set of predicates *Preds* and computes a set of formulas *ReachStates*$^\#$ that represents an over-approximation $\varphi^\#_{reach}$. Furthermore, AbstReach records its intermediate computation steps in a labeled tree *Parent*. (In the next section we will show how this tree can be used to discover new predicates when a refined abstraction is needed.)

The initialization steps of AbstReach are shown in lines 1–5 of Figure 1.2. First, we construct the abstraction function α according to Equation (1.16), and then use it to construct an over-approximation $post^\#$ of the post-condition function according to Equation (1.14). We initialize *ReachStates*$^\#$ with an over-approximation of the initial program states, which corresponds to the first disjunct in Equation (1.15). Since the initial states do not have any

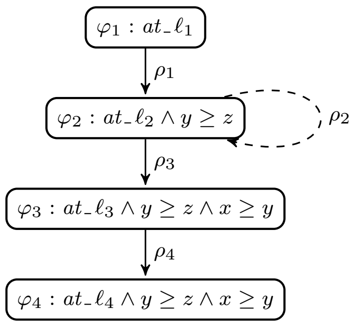
```
    **function** AbstReach
    **input**
      $Preds$ - predicates
    **begin**
1     $\alpha := \lambda\varphi \,.\, \bigwedge\{p \in Preds \mid \varphi \models p\}$
2     $post^{\#} := \lambda(\varphi, \rho) \,.\, \alpha(post(\varphi, \rho))$
3     $ReachStates^{\#} := \{\alpha(\varphi_{init})\}$
4     $Parent := \emptyset$
5     $Worklist := ReachStates^{\#}$
6     **while** $Worklist \neq \emptyset$ **do**
7       $\varphi :=$ choose from $Worklist$
8       $Worklist := Worklist \setminus \{\varphi\}$
9       **for** each $\rho \in \mathcal{T}$ **do**
10        $\varphi' := post^{\#}(\varphi, \rho)$
11        **if** $\varphi' \not\models \bigvee ReachStates^{\#}$ **then**
12          $ReachStates^{\#} := \{\varphi'\} \cup ReachStates^{\#}$
13          $Parent := \{(\varphi, \rho, \varphi')\} \cup Parent$
14          $Worklist := \{\varphi'\} \cup Worklist$
15    **return** $(ReachStates^{\#}, Parent)$
    **end**
```

Fig. 1.2 Algorithm AbstReach for abstract reachability computation wrt. a given finite set of predicates.

predecessors, *Parent* is initially empty. Finally, we create a worklist *Worklist* that contains sets of states on which $post^{\#}$ has not been applied yet.

The main part of AbstReach in lines 6–14 implements the iterative application of $post^{\#}$ in Equation (1.15) using a while loop. The loop termination condition checks if *Worklist* has any items to process. In case the worklist is not empty, we choose such an item, say φ, and remove it from the worklist. For brevity, we leave the selection procedure unspecified, but note that various strategies are possible, e.g., breadth- or depth-first search. Then, we apply $post^{\#}$ wrt. each program transition, say ρ, on φ. Let φ' be the result of such an application. We add φ' to $ReachStates^{\#}$ if φ' contains some program states that are not already contained in one of the formulas in $ReachStates^{\#}$. We formulate the above test as an entailment check between φ' and the disjunction of all formulas in $ReachStates^{\#}$. Often, there is a formula ψ in $ReachStates^{\#}$ such that $\varphi' \models \psi$. Otherwise, that φ is added to $ReachStates^{\#}$, we record that φ' was computed by applying ρ on φ by adding a tuple $(\varphi, \rho, \varphi')$ to *Parent*. Finally, φ' is put on the worklist.

The loop execution terminates after a finite number of steps, since the range of $post^{\#}$ is finite (and is of size 2^n where n is the size of *Preds*). The disjunction of formulas in $ReachStates^{\#}$ is logically equivalent to $\varphi^{\#}_{reach}$.


$\varphi_1 : at\_\ell_1$
ρ_1
$\varphi_2 : at\_\ell_2 \wedge y \geq z$
ρ_2
ρ_3
$\varphi_3 : at\_\ell_3 \wedge y \geq z \wedge x \geq y$
ρ_4
$\varphi_4 : at\_\ell_4 \wedge y \geq z \wedge x \geq y$

$$\begin{aligned}
&\varphi_1 = \alpha(\varphi_{init})\\
&\varphi_2 = post^{\#}(\varphi_1, \rho_1)\\
&post^{\#}(\varphi_2, \rho_2) \models \varphi_2\\
&\varphi_3 = post^{\#}(\varphi_2, \rho_3)\\
&\varphi_4 = post^{\#}(\varphi_3, \rho_4)
\end{aligned}$$

Fig. 1.3 Applying AbstReach on the program in Figure 1.1 and the set of predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq y\}$. The nodes $\varphi_1, \ldots, \varphi_4$ represent elements of $ReachStates^{\#}$. Labeled edges connecting the nodes represent *Parent*. The dotted edge denotes the entailment relation between $post^{\#}(\varphi_2, \rho_2)$ and φ_2.

Example 13. We describe the application of AbstReach on our example program when $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq y\}$. Figure 1.3 provides a pictorial illustration. Example 12 provides details on computed over-approximations of post-conditions.

After constructing α and $post^{\#}$ for the given predicates, we compute $\varphi_1 = (at\_\ell_1)$ and put it into $ReachStates^{\#}$ and into *Worklist*. See the node φ_1 in Figure 1.3.

During the first loop iteration, we choose φ_1 to be the element taken from the worklist. Now we compute $post^{\#}$ wrt. each program transition. For ρ_1 we obtain $\varphi_2 = (at\_\ell_2 \wedge y \geq z)$. The entailment check $\varphi_2 \models \bigvee ReachStates^{\#}$ fails, since $\bigvee ReachStates^{\#}$ is equal to φ_1 and $\varphi_2 \not\models \varphi_1$. Hence, φ_2 is added to $ReachStates^{\#}$. As a result, the tuple $(\varphi_1, \rho_1, \varphi_2)$ is added to *Parent* and φ_2 becomes a worklist item. See the node φ_2 as well as the edge between φ_1 and φ_2 in Figure 1.3. We continue with applying program transitions on φ_1. For ρ_2 we obtain $post^{\#}(\varphi_1, \rho_2) = false$. Since $false \models \bigvee ReachStates^{\#}$ there is no addition to $ReachStates^{\#}$. Similarly, applying $\rho_3, \ldots, \rho_5$ does not modify $ReachStates^{\#}$.

We start the second loop iteration with $ReachStates^{\#} = \{\varphi_1, \varphi_2\}$, $Worklist = \{\varphi_2\}$, and $Parent = \{(\varphi_1, \rho_1, \varphi_2)\}$. We choose φ_2 from the worklist. When applying $post^{\#}$ on φ_2 only ρ_2 and ρ_3 result sets of successor states that are not equal to *false*. We obtain $post^{\#}(\varphi_2, \rho_2) = (at\_\ell_2 \wedge y \geq z)$. Since $(at\_\ell_2 \wedge y \geq z)$ entails φ_2 and hence $\bigvee ReachStates^{\#}$, nothing is added to $ReachStates^{\#}$ and we proceed directly with ρ_3. For $\varphi_3 = post^{\#}(\varphi_2, \rho_3) = (at\_\ell_3 \wedge y \geq z \wedge x \geq y)$ we observe that $\varphi_3 \not\models \bigvee ReachStates^{\#}$. Hence, we add

φ_3 to *ReachStates*$^{\#}$ and *Worklist*, while $(\varphi_2, \rho_3, \varphi_3)$ is recorded in *Parent*. See the node φ_3 as well as the edge between φ_2 and φ_3 in Figure 1.3.

At the beginning of the third loop iteration we have *ReachStates*$^{\#}$ $= \{\varphi_1, \varphi_2, \varphi_3\}$, *Worklist* $= \{\varphi_3\}$, and *Parent* $= \{(\varphi_1, \rho_1, \varphi_2), (\varphi_2, \rho_3, \varphi_3)\}$. We choose φ_3 from the worklist. After computing φ_4 by applying ρ_4 and discovering that $\varphi_4 \not\models \bigvee$ *ReachStates*$^{\#}$, we add φ_4 following the algorithm. See the node φ_4 as well as the edge between φ_3 and φ_4 in Figure 1.3. Since all other program transition yield *false* we proceed with the next iteration.

The fourth loop iteration removes φ_4 from the worklist, but does not add any new elements to it. Hence AbstReach terminates and outputs *ReachStates*$^{\#}$ $= \{\varphi_1, \ldots, \varphi_4\}$ as well as *Parent* $= \{(\varphi_1, \rho_1, \varphi_2), (\varphi_2, \rho_3, \varphi_3), (\varphi_3, \rho_4, \varphi_4)\}$. □

Our algorithm is presented as an instance of *abstract interpretation* [23]. This presentation is very general. It allows one to leave open many design choices. For example, the algorithm may be split into two steps. The first ("offline") step is to compute the function *post*$^{\#}$. The second step is to iterate *post*$^{\#}$ until a fixpoint is reached. Often, e.g., in [2, 3, 5, 16], conjunctions of predicates are viewed as *abstract states* (which can possibly be represented as bitvectors). Instead of constructing the function *post*$^{\#}$ directly, one first constructs a relation between abstract states. If one views this relation as the transition relation of an *abstract program* $\mathcal{P}^{\#}$ (the "abstraction of the program $\mathcal{P}$"), then the abstraction of the post operator for the program $\mathcal{P}$ can be phrased as the post operator of the (finite-state) abstract program $\mathcal{P}^{\#}$,

$$post^{\#}_{\mathcal{P}} = post_{\mathcal{P}^{\#}}.$$

In this view, the first step is phrased as the construction of a finite model and the second step is phrased as model checking (see also Chapters 12 and 18).

1.5.2 Termination and transition predicate abstraction

In this section, we show how predicate abstraction can be used for computing transition invariants, and thus proving program termination.

In principle, transition invariants can be computed by applying the iterative scheme in Equation (1.10) and then restricting the obtained result to reachable states by relying on Equation (1.11). The iteration of *comp* finishes when no new pair of program states is discovered. Unfortunately, such an iteration process does not terminate in finite time, for similar reasons as presented in Section 1.5.1.

Instead of computing ψ_{ti} we compute its over-approximation by a superset $\psi^{\#}_{ti}$. Then, we check whether $\psi^{\#}_{ti}$ is disjunctively well-founded. If $\psi^{\#}_{ti}$ satisfies disjunctive well-foundedness condition then ψ_{ti} is disjunctively well-founded as well. Hence the program terminates.

Similarly to the computation of ψ_{ti}, we compute $\psi^{\#}_{ti}$ by applying iteration. However, instead of iteratively applying the relational composition function *comp* we use its over-approximation $\mathit{comp}^{\#}$ such that

$$\forall \psi : \mathit{comp}(\psi) \models \mathit{comp}^{\#}(\psi) \ . \tag{1.17}$$

We decompose the computation of $\mathit{comp}^{\#}$ into two steps. First, we apply *comp* and then we over-approximate the result using a function $\ddot{\alpha}$ such that

$$\forall \psi : \psi \models \ddot{\alpha}(\psi) \ . \tag{1.18}$$

That is, given an over-approximating function $\ddot{\alpha}$ we define $\mathit{comp}^{\#}$ as follows.

$$\mathit{comp}^{\#}(\psi) \ = \ \ddot{\alpha}(\mathit{comp}(\psi)) \tag{1.19}$$

Finally, we can obtain $\psi^{\#}_{ti}$ by using a previously computed over-approximation of reachable states $\varphi^{\#}_{\mathit{reach}}$ as follows.

$$\begin{aligned} \psi^{\#}_{ti} &= \mathit{comp}^{\#}(\varphi^{\#}_{\mathit{reach}} \land V' = V) \lor \\ &\quad\ \mathit{comp}^{\#}(\mathit{comp}^{\#}(\varphi^{\#}_{\mathit{reach}} \land V' = V)) \lor \dots \\ &= \textstyle\bigvee_{i \geq 1} (\mathit{comp}^{\#})^{i}(\varphi^{\#}_{\mathit{reach}} \land V' = V) \end{aligned} \tag{1.20}$$

We formalize our over-approximation based transition invariant computation as follows. The strongest transition invariant of the program is contained in the result of abstract computation given by Equation (1.20). Formally, $\psi_{ti} \models \psi^{\#}_{ti}$

Transition predicate abstraction We construct an over-approximation $\psi^{\#}_{ti}$ using a given set of building blocks, so-called *transition predicates*. Each transition predicate is a formula over the program variables V and their primed verions V', which represents a binary relation over program states [53].

We fix a finite set of *transition predicates* $\mathit{TransPreds} = \{\ddot{p}_1, \dots, \ddot{p}_n\}$. Then, we define an over-approximation of ψ that is constructed using *TransPreds* as follows.

$$\ddot{\alpha}(\psi) = \bigwedge \{\ddot{p} \in \mathit{TransPreds} \mid \psi \models \ddot{p}\} \tag{1.21}$$

If the set of entailed transition predicates is empty then the result of applying transition predicate abstraction is $\bigwedge \emptyset$, which is equivalent to *true*.

Example 14. For example, we consider a set of predicates $\mathit{TransPreds} = \{\mathit{at\_}\ell_1, \dots, \mathit{at\_}\ell_5, \mathit{at'\_}\ell_1, \dots, \mathit{at'\_}\ell_5, x \geq 0, x' \geq x+1, x' \geq x, y' \geq y, y' \geq$

$y+1\}$. We will apply transition predicate abstraction on the transition ρ_2 in the program shown in Figure 1.1. We compute $\ddot{\alpha}(\mathit{goto}(\ell_2, \ell_2) \wedge x+1 \leq y \wedge x' = x+1 \wedge \mathit{unchanged}(y,z))$ as follows. First, we check the logical consequence between the argument to the abstraction function and each of the predicates and obtain the following set of entailed predicates:

$$\{\mathit{at\_}\ell_2, \mathit{at'\_}\ell_2, x \geq 0, x' \geq x+1, x' \geq x, y' \geq y\} \ .$$

Then, we take the conjunction of the entailed predicates as the result of the abstraction.

$$\begin{aligned} &\bigwedge\{\mathit{at\_}\ell_2, \mathit{at'\_}\ell_2, x \geq 0, x' \geq x+1, x' \geq x, y' \geq y\} \\ = \ &\mathit{at\_}\ell_2 \wedge \mathit{at'\_}\ell_2 \wedge x \geq 0 \wedge x' \geq x+1 \wedge y' \geq y \end{aligned}$$

□

The transition predicate abstraction function in Equation (1.21) approximates ψ using a conjunction of transition predicates, which requires n entailment checks where n is the number of given transition predicates.

Example 15. We use transition predicate abstraction to compute $\psi^{\#}_{ti}$ for our example program from Figure 1.1 following the iterative scheme presented in Equation (1.20). Since we are interested in proving termination we will ignore the assertion statement occuring in the program. As a consequence, in this example will not take transitions ρ_4 and ρ_5 into account.

For simplicity, we use $\varphi^{\#}_{\mathit{reach}} = \mathit{true}$, which states that the set of reachable states is contained in the set of all possible program states represented by true. Let $\mathit{TransPreds} = \{\mathit{false}, \mathit{at\_}\ell_1, \dots, \mathit{at\_}\ell_3, \mathit{at'\_}\ell_1, \dots, \mathit{at'\_}\ell_3, x \leq y, y'-x' \leq y-x-1\}$.

First, we compute the first step of the over-approximation $\mathit{comp}^{\#}(\varphi^{\#}_{\mathit{reach}} \wedge V' = V)$:

$$\begin{aligned} \mathit{comp}^{\#}(\varphi^{\#}_{\mathit{reach}} \wedge V' = V) &= \ddot{\alpha}((\varphi^{\#}_{\mathit{reach}} \wedge V' = V) \circ \rho_{\mathcal{P}}) \\ &= \ddot{\alpha}((\varphi^{\#}_{\mathit{reach}} \wedge V' = V) \circ (\rho_1 \vee \rho_2 \vee \rho_3)) \\ &= \ddot{\alpha}(\rho_1 \vee \rho_2 \vee \rho_3) \end{aligned}$$

We obtain the following abstractions for each of the transitions:

$$\begin{aligned} \psi_1 &= \ddot{\alpha}(\rho_1) = \mathit{goto}(\ell_1, \ell_2) \\ \psi_2 &= \ddot{\alpha}(\rho_2) = (\mathit{goto}(\ell_2, \ell_2) \wedge x \leq y \wedge y' - x' \leq y - x - 1) \\ \psi_3 &= \ddot{\alpha}(\rho_3) = \mathit{goto}(\ell_2, \ell_3) \end{aligned}$$

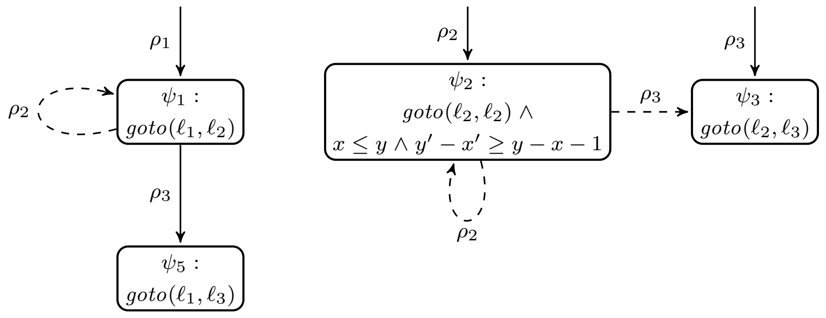


Fig. 1.4 Abstract transitions computed in Example 15. Solid edges connecting the nodes represent how nodes were computed. Dotted edges denote entailment relation.

We apply $comp^\#$ on $\psi_1, \dots, \psi_3$ and obtain the following non-empty abstractions.

$$\psi_4 = \ddot{\alpha}(\psi_1 \circ \rho_2) = goto(\ell_1, \ell_2)$$
$$\psi_5 = \ddot{\alpha}(\psi_1 \circ \rho_3) = goto(\ell_1, \ell_3)$$
$$\psi_6 = \ddot{\alpha}(\psi_2 \circ \rho_2) = (goto(\ell_2, \ell_2) \wedge x \leq y \wedge y' - x' \geq y - x - 1)$$
$$\psi_7 = \ddot{\alpha}(\psi_2 \circ \rho_3) = (goto(\ell_2, \ell_3) \wedge x \leq y \wedge y' - x' \geq y - x - 1)$$

We observe that $\psi_4 \models \psi_1$, $\psi_6 \models \psi_2$, and $\psi_7 \models \psi_3$, hence, ψ_4, ψ_6, and ψ_7 can be ignored. We apply $comp^\#$ on ψ_5 and ψ_7 and observe that the resulting abstractions are empty. Hence, we finish the iterative computation and obtain

$$\psi_{ti}^\# \;=\; \psi_1 \vee \psi_2 \vee \psi_3 \vee \psi_5\ .$$

The computed transition invariant $\psi_{ti}^\#$ is disjunctively well-founded. Each disjunct in $\psi_{ti}^\#$ whose start and finish locations are not equal, e.g., ψ_1 with the start location ℓ_1 and finish location ℓ_2, is well-founded. The remaining disjunct ψ_2 is well-founded since the value of $y - x$ is greater than zero whenever ψ_2 makes a step and decreases during each step of ψ_2. Hence, we conclude that $\psi_{ti}^\#$ is contained in a finite union of well-founded relations. Thus, the program terminates.

We represent the above computation pictorially in Figure 1.4. $\square$

Algorithm TransAbstReach We combine the characterization of abstract transition invariants using Equation (1.20) with the transition predicate abstraction function given in Equation (1.21) and obtain an algorithm

```
    function TransAbstReach
    input
      $ReachStates^{\#}$ - reachable abstract states
      $TransPreds$ - transition predicates
    begin
1     $\ddot{\alpha} := \lambda\ddot{\varphi} \,.\, \bigwedge\{\ddot{p} \in TransPreds \mid \ddot{\varphi} \models \ddot{p}\}$
2     $ReachTrans^{\#} := \emptyset$
3     $TransParent := \emptyset$
4     $Worklist := \{\varphi \wedge unchanged(V) \mid \varphi \in ReachStates^{\#}\}$
5     while $Worklist \neq \emptyset$ do
6       $\ddot{\varphi}$ := choose from $Worklist$
7       $Worklist := Worklist \setminus \{\ddot{\varphi}\}$
8       for each $\rho \in \mathcal{T}$ do
9         $\ddot{\varphi}' := \ddot{\alpha}(\ddot{\varphi} \circ \rho)$
10        if $\neg(\exists \ddot{\psi} \in ReachTrans^{\#} : \ddot{\varphi}' \models \ddot{\psi})$ then
11          $ReachTrans^{\#} := \{\ddot{\varphi}\} \cup ReachTrans^{\#}$
12          $TransParent := \{(\ddot{\varphi}, \rho, \ddot{\varphi}')\} \cup TransParent$
13          $Worklist := \{\ddot{\varphi}'\} \cup Worklist$
14    return $(ReachTrans^{\#}, TransParent)$
    end
```

Fig. 1.5 Abstract transitive closure computation.

TransAbstReach for computing *ReachTrans*$^{\#}$. The algorithm is shown in Figure 1.5.

TransAbstReach takes as input a finite set of transition predicates *TransPreds* and computes a set of formulas *ReachTrans*$^{\#}$ that represents an over-approximation $\psi^{\#}_{ti}$. Furthermore, TransAbstReach records its intermediate computation steps in a labeled tree *TransParent*. In the next section we will show how this tree can be used to discover new transition predicates when a refined abstraction is needed.

The initialization steps of TransAbstReach are shown in lines 1–4 in Figure 1.5. First, we construct the abstraction function $\ddot{\alpha}$ according to Equation (1.21), and then use it to construct an over-approximation of the relational composition with program transitions. *ReachTrans*$^{\#}$ is initially empty, which corresponds to the fact that we are interested in the irreflexive transitive closure following Equation (1.20). Since the initial relations do not have any predecessors, *TransParent* is initially empty. Finally, we initialize the worklist *Worklist* with a set of identity relations over program states that are restricted to the sets of states represented by elements of *ReachStates*$^{\#}$. Such initalization together with the first iteration of the while loop correspond to the first disjunct in Equation (1.20).

The main part of AbstReach in lines 5–13 implements the iterative application of *comp*$^{\#}$ in Equation (1.20) using a while loop. Since we are interested in applying individual program transitions one by one, we rely on a direct application of relational composition $\circ$ and transition predicate abstraction $\ddot{\alpha}$. The loop termination condition checks if *Worklist* has any items to process. In case the worklist is not empty, we choose such an item, say $\ddot{\varphi}$, and remove it from the worklist. For brevity, we leave the selection procedure unspecified, but note that various strategies are possible, e.g., breadth- or depth-first search. Then, we apply $\circ$ and $\ddot{\alpha}$ wrt. each program transition, say ρ, on $\ddot{\varphi}$. Let $\ddot{\varphi}'$ be the result of such an application. We add $\ddot{\varphi}'$ to $\mathit{ReachTrans}^{\#}$ if $\ddot{\varphi}'$ contains some pairs of program states that are not already contained in one of the formulas in $\mathit{ReachTrans}^{\#}$. We formulate the above test as an entailment check between $\ddot{\varphi}'$ and the disjunction of all formulas in $\mathit{ReachTrans}^{\#}$. Often, there is a formula $\ddot{\psi}$ in $\mathit{ReachStates}^{\#}$ such that $\ddot{\varphi}' \models \ddot{\psi}$. Otherwise, $\ddot{\varphi}$ is added to $\mathit{ReachTrans}^{\#}$, we record that $\ddot{\varphi}'$ was computed by applying ρ on $\ddot{\varphi}$ by adding a tuple $(\ddot{\varphi}, \rho, \ddot{\varphi}')$ to *TransParent*. Finally, $\ddot{\varphi}'$ is put on the worklist.

The loop execution terminates after a finite number of steps, since the range of $\ddot{\alpha}$ is finite (and is of size 2^n where n is the size of *TransPreds*). The disjunction of formulas in $\mathit{ReachTrans}^{\#}$ is logically equivalent to $\psi_{ti}^{\#}$.

Example 16. We illustrate TransAbstReach by showing how it automates the computation presented in Example 15. We again consider our example program from Figure 1.1 together with a set of transition predicates $\mathit{TransPreds} = \{\mathit{false}, \mathit{at}\_\ell_1, \dots, \mathit{at}\_\ell_3, \mathit{at}'\_\ell_1, \dots, \mathit{at}'\_\ell_3\}$ and an over-approximation of reachable states $\mathit{ReachStates}^{\#} = \{\mathit{true}\}$. After executing the initialization steps in TransAbstReach we obtain $\mathit{ReachTrans}^{\#} = \emptyset$, $\mathit{TransParent} = \emptyset$, and $\mathit{Worklist} = \{\mathit{true} \wedge \mathit{unchanged}(x, y, z)\}$.

The first iteration of the while loop chooses $\ddot{\varphi} = (\mathit{true} \wedge \mathit{unchanged}(x, y, z))$ and removes it from *Worklist*. Now TransAbstReach iterates through the transitions of the program. First, we consider $\rho = \rho_1$ and obtain

$$\ddot{\varphi}' = \ddot{\alpha}((\mathit{true} \wedge \mathit{unchanged}(x, y, z)) \circ \rho_1) = \mathit{goto}(\ell_1, \ell_2)\ .$$

Since $\mathit{ReachTrans}^{\#} = \emptyset$, we obtain

$$\begin{aligned}
\mathit{ReachTrans}^{\#} &= \{\mathit{goto}(\ell_1, \ell_2)\} \\
\mathit{TransParent} &= \{(\mathit{true} \wedge \mathit{unchanged}(x, y, z), \rho_1, \mathit{goto}(\ell_1, \ell_2))\} \\
\mathit{Worklist} &= \{\mathit{goto}(\ell_1, \ell_2)\}
\end{aligned}$$

and proceed with the remaing program transitions. For $\rho = \rho_2$ we obtain $\ddot{\varphi}' = \mathit{goto}(\ell_2, \ell_2)$. Since $\ddot{\varphi}'$ does not entail the transition relation already contained in $\mathit{ReachTrans}^{\#}$, we obtain (in this example, "…" denotes previously assigned value)

$$\begin{aligned}
\mathit{ReachTrans}^{\#} &= \{\mathit{goto}(\ell_2,\ell_2),\dots\}\\
\mathit{TransParent} &= \{(\mathit{true} \wedge \mathit{unchanged}(x,y,z),\rho_2,\mathit{goto}(\ell_2,\ell_2),\dots\}\\
\mathit{Worklist} &= \{\mathit{goto}(\ell_2,\ell_2),\mathit{goto}(\ell_1,\ell_2)\}
\end{aligned}$$

After applying $\rho = \rho_3$ we obtain $\ddot{\varphi}' = \mathit{goto}(\ell_2,\ell_3)$ that leads to

$$\begin{aligned}
\mathit{ReachTrans}^{\#} &= \{\mathit{goto}(\ell_2,\ell_3),\dots\}\\
\mathit{TransParent} &= \{(\mathit{true} \wedge \mathit{unchanged}(x,y,z),\rho_3,\mathit{goto}(\ell_2,\ell_3)),\dots\}\\
\mathit{Worklist} &= \{\mathit{goto}(\ell_2,\ell_3),\mathit{goto}(\ell_2,\ell_2),\mathit{goto}(\ell_1,\ell_2)\}
\end{aligned}$$

Now, we proceed with the second iteration of the while loop. We choose $\ddot{\varphi} = \mathit{goto}(\ell_1,\ell_2)$ and proceed with applying program transitions. Applying ρ_1 yields $\ddot{\varphi}' = \mathit{false}$. For $\rho = \rho_2$ we obtain $\ddot{\varphi} = \mathit{goto}(\ell_1,\ell_2)$. Since there exists an element of $\mathit{ReachTrans}^{\#}$ that is entailed by $\ddot{\varphi}'$, namely $\mathit{goto}(\ell_1,\ell_2)$, the computed $\ddot{\varphi}'$ is discarded. Applying ρ_3 yields $\ddot{\varphi}' = \mathit{goto}(\ell_1,\ell_3)$ and leads to

$$\begin{aligned}
\mathit{ReachTrans}^{\#} &= \{\mathit{goto}(\ell_1,\ell_3),\dots\}\\
\mathit{TransParent} &= \{(\mathit{goto}(\ell_1,\ell_2),\rho_3,\mathit{goto}(\ell_1,\ell_3)),\dots\}\\
\mathit{Worklist} &= \{\mathit{goto}(\ell_1,\ell_3),\mathit{goto}(\ell_2,\ell_3),\mathit{goto}(\ell_2,\ell_2)\}
\end{aligned}$$

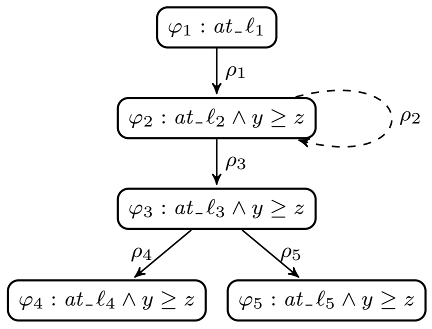
Subsequent iterations of the while loop proceed similarly and modify neither $\mathit{ReachTrans}^{\#}$ nor $\mathit{TransParent}$. Finally, $\mathit{Worklist}$ becomes empty and TransAbstReach terminates. We obtain the following output:

$$\begin{aligned}
\mathit{ReachTrans}^{\#} &= \{\mathit{goto}(\ell_1,\ell_3),\mathit{goto}(\ell_2,\ell_3),\mathit{goto}(\ell_2,\ell_2),\mathit{goto}(\ell_1,\ell_2)\}\\
\mathit{TransParent} &= \{(\mathit{goto}(\ell_1,\ell_2),\rho_3,\mathit{goto}(\ell_1,\ell_3)),\\
&\qquad (\mathit{true} \wedge \mathit{unchanged}(x,y,z),\rho_3,\mathit{goto}(\ell_2,\ell_3)),\\
&\qquad (\mathit{true} \wedge \mathit{unchanged}(x,y,z),\rho_2,\mathit{goto}(\ell_2,\ell_2)),\\
&\qquad (\mathit{true} \wedge \mathit{unchanged}(x,y,z),\rho_1,\mathit{goto}(\ell_1,\ell_2))\}
\end{aligned}$$

□

1.6 Abstraction refinement

The algorithm AbstReach requires a set of predicates in order to compute an over-approximation of the set of reachable program states. Similarly, the algorithm TransAbstReach requires a set of transition predicates in order


$\varphi_1 : at\_\ell_1$
ρ_1
$\varphi_2 : at\_\ell_2 \wedge y \geq z$
ρ_2
ρ_3
$\varphi_3 : at\_\ell_3 \wedge y \geq z$
ρ_4
ρ_5
$\varphi_4 : at\_\ell_4 \wedge y \geq z$
$\varphi_5 : at\_\ell_5 \wedge y \geq z$

$$\begin{aligned}
&\varphi_1 = \alpha(\varphi_{init})\\
&\varphi_2 = post^{\#}(\varphi_1, \rho_1)\\
&post^{\#}(\varphi_2, \rho_2) \models \varphi_2\\
&\varphi_3 = post^{\#}(\varphi_2, \rho_3)\\
&\varphi_4 = post^{\#}(\varphi_3, \rho_4)\\
&\varphi_5 = post^{\#}(\varphi_3, \rho_5)
\end{aligned}$$

Fig. 1.6 Abstract reachability computation with $\mathit{Preds} = \{\mathit{false}, at\_\ell_1, \dots, at\_\ell_5, y \geq z\}$.

to compute an over-approximation of the set of reachable pairs of program states. Finding the right set of predicates (or of transition predicates) that yields a sufficiently precise over-approximation is a difficult task.

1.6.1 Refinement of predicate abstraction

The procedure for refining predicate abstraction considers certain program paths as the main source of information. By exploring such paths we can obtain an adequate set of predicates to prove the program correct.

Analysis of counterexample paths We start with an example that illustrates the impact of over-approximatioo and how it can be eliminated.

Example 17. In Example 13, the provided set of predicates was adequate for proving program safety. Omitting just one predicate, e.g., provide the predicates $\mathit{Preds} = \{\mathit{false}, at\_\ell_1, \dots, at\_\ell_5, y \geq z\}$ without $x \geq y$ leads to an over-approximation $\varphi^{\#}_{reach}$ that has a non-empty intersection with the error states. As shown in Figure 1.6, we have $\varphi_5 \wedge \varphi_{err} \not\models \mathit{false}$. That is, AbstReach fails to prove the property without the predicate $x \geq y$.

We analyse the reason for the excessive over-approximation. Figure 1.6 shows that the *Parent* relation records a sequence of three steps leading to the computation of φ_5. First, we apply ρ_1 to φ_1 and compute φ_2. Then, φ_3 is obtained by applying ρ_3 to φ_2. Finally, ρ_5 is applied to φ_3 and results in φ_5. Thus, we note that the sequence of program transitions ρ_1, ρ_3, and ρ_5 determined φ_5. We refer to this sequence as a counterexample path. Using this path and the functions α and $post^{\#}$ corresponding to the current set of predicates we obtain

$$\varphi_5 = post^{\#}(post^{\#}(post^{\#}(\alpha(\varphi_{init}), \rho_1), \rho_3), \rho_5)\ .$$

That is, φ_5 is equal to the over-approximation of the post-condition computed along the counterexample path.

Now we check if the counterexample path also leads to the error states when no over-approximation is applied. First we compute

$$\begin{aligned} post(post(post(\varphi_{init}, \rho_1), \rho_3), \rho_5) &= post(post(at\_\ell_2 \land y \geq z, \rho_3), \rho_5) \\ &= post(at\_\ell_3 \land y \geq z \land x \geq y, \rho_5) \\ &= false\ . \end{aligned}$$

Hence, by executing the program transitions ρ_1, ρ_3, and ρ_5 it is not possible to reach any error state. We conclude that the over-approximation is too coarse, at least when dealing with the above path.

We need a more precise over-approximation that will prevent $post^{\#}$ from including states that lead to error states along the path ρ_1, ρ_3, and ρ_5. Concretely, we need a refined abstraction function α and a corresponding $post^{\#}$ such that the execution of AbstReach along the counterexample path does not compute a set of states that contains some error states:

$$post^{\#}(post^{\#}(post^{\#}(\alpha(\varphi_{init}), \rho_1), \rho_3), \rho_5) \land \varphi_{err} \models false\ .$$

We consider the intermediate steps of the above condition and define sets of states $\psi_1, \ldots, \psi_4$ that provide an adequate over-approximation along the path as follows.

$$\begin{aligned} &\varphi_{init} \models \psi_1 \\ &post(\psi_1, \rho_1) \models \psi_2 \\ &post(\psi_2, \rho_3) \models \psi_3 \\ &post(\psi_3, \rho_5) \models \psi_4 \\ &\psi_4 \land \varphi_{err} \models false \end{aligned}$$

The over-approximation given by $\psi_1, \ldots, \psi_4$ is adequate since it guarantees that no error state is reached, while still allowing additional states to be reachable. For example, we consider the following solution to the above condition.

ψ_1	ψ_2	ψ_3	ψ_4
at_ℓ_1	$at\_\ell_2 \land y \geq z$	$at\_\ell_3 \land x \geq z$	$false$

```
    function MakePath
    input
      $\psi$ - reachable abstract state
      $Parent$ - predecessor relation
    begin
1     $path$ := empty sequence
2     $\varphi'$ := $\psi$
3     while exist $\varphi$ and $\rho$ such that $(\varphi, \rho, \varphi') \in Parent$ do
4        $path$ := $\rho$ . $path$
5        $\varphi'$ := $\varphi$
6     return $path$
    end
```

Fig. 1.7 Path computation.

We can use the obtained solution to refine α in the following way. By adding $\psi_1, \ldots, \psi_4$ to the set of predicates *Preds* we guarantee that the resulting α and $post^{\#}$ are sufficiently precise to show that no error state is reachable along the path ρ_1, ρ_3, and ρ_5. Formally, we obtain

$$\begin{aligned}
&\alpha(\varphi_{init}) \models \psi_1 \\
&post^{\#}(\psi_1, \rho_1) \models \psi_2 \\
&post^{\#}(\psi_2, \rho_3) \models \psi_3 \\
&post^{\#}(\psi_3, \rho_5) \models \psi_4 \\
&\psi_4 \wedge \varphi_{err} \models false
\end{aligned}$$

□

We put the above approach for analyzing counterexample compute by AbstReach into algorithms MakePath, FeasiblePath, and RefinePath.

The algorithm MakePath is shown in Figure 1.7. It takes as input a reachable abstract state ψ together with a *Parent* relation. We view *Parent* as a tree where ψ occurs as a node. MakePath outputs a sequence of program transitions that labels the tree edges connecting ψ with the root of the tree. The sequence is constructed iteratively by a backward traversal starting from the input node. The variable *path* keep track of the construction.

Example 18. For our example tree in Figure 1.6 we construct the path by making a call MakePath$(\varphi_5, Parent)$. Then, *path* is extended with the transitions ρ_5, ρ_3, and ρ_1 by considering the edges $(\varphi_3, \rho_5, \varphi_5)$, $(\varphi_2, \rho_3, \varphi_3)$, and $(\varphi_1, \rho_1, \varphi_2)$, respectively. Finally, $path = \rho_1\rho_3\rho_5$ is returned as output. □

```
    function FEASIBLEPATH
    input
      $\rho_1 \ldots \rho_n$ - path
    begin
1     $\varphi \ := \ post(\varphi_{init}, \rho_1 \circ \ldots \circ \rho_n)$
2     if $\varphi \land \varphi_{err} \not\models false$ then
3         return $true$
4     else
5         return $false$
    end
```

Fig. 1.8 Feasibility of a path.

```
    function REFINEPATH
    input
      $\rho_1 \ldots \rho_n$ - path
    begin
1     $\varphi_0, \ldots, \varphi_n \ :=$ compute such that
2         $(\varphi_{init} \models \varphi_0) \land$
3         $(post(\varphi_0, \rho_1) \models \varphi_1) \land \ldots \land (post(\varphi_{n-1}, \rho_n) \models \varphi_n) \land$
4         $(\varphi_n \land \varphi_{err} \models false)$
5     return $\{\varphi_0, \ldots, \varphi_n\}$
    end
```

Fig. 1.9 Counterexample guided discovery of predicates.

The algorithm FEASIBLEPATH is shown in Figure 1.8. It takes as input a sequence of program transitions $\rho_1 \ldots \rho_n$ and checks if there is a computation that is produced by this sequence. The check uses the post-condition function and the relational composition of transitions.

Example 19. When applying FEASIBLEPATH on our example path $\rho_1\rho_3\rho_5$ we obtain the following intermediate results. First, the relational composition of transitions yields

$$\rho_1 \circ \rho_3 \circ \rho_5 = false \ .$$

Hence, FEASIBLEPATH sets φ to *false* and then returns *false*. □

The algorithm REFINEPATH is shown in Figure 1.9. It takes as input a sequence of program transitions $\rho_1 \ldots \rho_n$ and computes sets of states $\varphi_0, \ldots, \varphi_n$ satisfying the following conditions. First, we have $\varphi_{init} \models \varphi_0$ and $\varphi_n \land \varphi_{err} \models false$. Then, for each $i \in 1..n$ we obtain $post(\varphi_{i-1}, \rho_i) \models \varphi_i$. Thus, $\varphi_0, \ldots, \varphi_n$ computed by REFINEPATH can be used for refining predicate abstraction. If $\varphi_0, \ldots, \varphi_n$ are added to *Preds* then the resulting α and

```
     function AbstRefineLoop
     begin
1      $Preds := \emptyset$
2      repeat
3        $(ReachStates^{\#}, Parent) := \textsc{AbstReach}(Preds)$
4        if exists $\psi \in ReachStates^{\#}$ such that $\psi \wedge \varphi_{err} \not\models false$ then
5          $path := \textsc{MakePath}(\psi, Parent)$
6          if $\textsc{FeasiblePath}(path)$ then
7            return "counterexample path: $path$ "
8          else
9            $Preds := \textsc{RefinePath}(path) \cup Preds$
10       else
11         return "program is correct"
     end.
```

Fig. 1.10 Predicate abstraction and refinement loop.

post$^{\#}$ guarantee that the following conditions hold.

$$\begin{aligned}
&\alpha(\varphi_{init}) \models \varphi_0\\
&post^{\#}(\varphi_0, \rho_1) \models \varphi_1\\
&\dots\\
&post^{\#}(\varphi_{n-1}, \rho_n) \models \varphi_n\\
&\varphi_n \wedge \varphi_{err} \models false
\end{aligned}$$

Here, we omit details of a particular algorithm for finding $\varphi_0, \dots, \varphi_n$ that satisfy the above conditions. We discuss possible alternatives in Section 1.7.

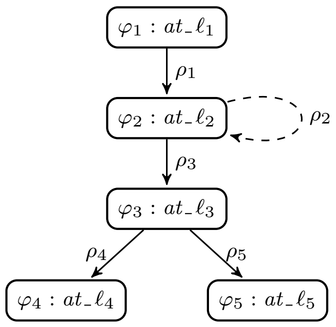
Example 20. As discussed in Example 17, the application of RefinePath on $\rho_1\rho_3\rho_5$ yields a sequence of sets of states that can refine the abstraction to become sufficiently precise at least for dealing with the considered path. □

In our high-level presentation of the algorithm, we leave open many issues for optimization. In particular, line 3 of AbstRefineLoop is often improved by an incremental procedure called *lazy abstraction* [27, 39].

Algorithm for counterexample guided abstraction refinement

We put together the building blocks described in the previous section into an algorithm AbstRefineLoop that verifies reachability properties using predicate abstraction and its counterexample guided refinement. See Figure 1.10.

Given a program, AbstRefineLoop discovers a proof or a counterexample by repeatedly applying the following steps. First, we compute an over-approximation $\varphi^{\#}_{reach}$ of the set of reachable states using an abstraction function defined wrt. the set of predicates *Preds*, which is empty initially. The



$$\begin{aligned}
&\varphi_1 = \alpha(\varphi_{init})\\
&\varphi_2 = post^{\#}(\varphi_1, \rho_1)\\
&post^{\#}(\varphi_2, \rho_2) \models \varphi_2\\
&\varphi_3 = post^{\#}(\varphi_2, \rho_3)\\
&\varphi_4 = post^{\#}(\varphi_3, \rho_4)\\
&\varphi_5 = post^{\#}(\varphi_3, \rho_5)
\end{aligned}$$

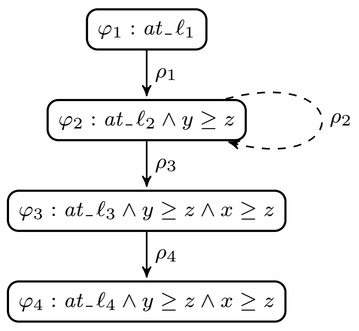
Fig. 1.11 Abstract reachability computation with $Preds = \{false, at\_\ell_1, \dots, at\_\ell_5\}$.

over-approximation $\varphi^{\#}_{reach}$ is represented by a set of formulas *ReachStates*$^{\#}$, where each formula represents a set of states. If the set of error states is disjoint from the computed over-approximation, then AbstRefineLoop stops the iteration process and reports that the program is correct. Otherwise, we consider a formula ψ in *ReachStates*$^{\#}$ that witnesses the intersection with the error states and use ψ in an attempt to refine the abstraction. Refinement is only possible if the discovered intersection is caused by the imprecision of the currently applied abstraction function. We clarify this question by first constructing a sequence of program transitions that was traversed during the computation of ψ. This sequence, called *path*, is analyzed using FeasiblePath. If there is a program computation that follows *path*, then AbstRefineLoop stops the iteration and reports that *path* is a counterexample. In case *path* is not feasible, we compute a set of predicates that refines the abstraction function by applying an algorithm RefinePath on *path*.

We observe that AbstRefineLoop never analyzes the same counterexample twice, i.e., the abstraction refinement process using RefinePath makes progress at each iteration.

Example 21. We illustrate AbstRefineLoop using our example program from Figure 1.1. To make the illustration more vivid, we assume that $Preds = \{false, at\_\ell_1, \dots, at\_\ell_5\}$ is the initial set of predicates, i.e., we anticipate that for proving our example correct we need to keep track of the program counter.

We start the first iteration by applying *ReachStates*$^{\#}$. The result is the set of formulas *ReachStates*$^{\#}$ connected by the relation *Parent* as shown in Figure 1.11. In this figure, *Parent* is denoted by solid arrows that connect the formulas. We observe that φ_5 has a non-empty intersection with φ_{err}, hence we proceed by setting ψ to φ_5. By applying MakePath we obtain $path = \rho_1\rho_3\rho_5$. At the next step, FeasiblePath reports that this path is not feasible, hence we proceed with the abstraction refinement. RefinePath discovers


$\varphi_1 : at\_\ell_1$
ρ_1
$\varphi_2 : at\_\ell_2 \wedge y \geq z$
ρ_2
ρ_3
$\varphi_3 : at\_\ell_3 \wedge y \geq z \wedge x \geq z$
ρ_4
$\varphi_4 : at\_\ell_4 \wedge y \geq z \wedge x \geq z$

$$\begin{aligned}
&\varphi_1 = \alpha(\varphi_{init})\\
&\varphi_2 = post^{\#}(\varphi_1, \rho_1)\\
&post^{\#}(\varphi_2, \rho_2) \models \varphi_2\\
&\varphi_3 = post^{\#}(\varphi_2, \rho_3)\\
&\varphi_4 = post^{\#}(\varphi_3, \rho_4)
\end{aligned}$$

Fig. 1.12 Applying AbstReach on the program in Figure 1.1 and the set of predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq z\}$.

that the predicates $y \geq z$ and $x \geq z$ are sufficient to refine the abstraction such that *path* no longer leads to an error state even under abstraction.

We start the second iteration of AbstRefineLoop with the new set of predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq z\}$, which contains the predicates that were discovered during the first iteration. See Figure 1.12 for the obtained set $ReachStates^{\#}$ and relation *Parent*. We observe that each formula in $ReachStates^{\#}$ has an empty intersection with φ_{err}, hence AbstRefineLoop reports that the program is correct. □

1.6.2 Refinement of transition predicate abstraction

The algorithm TransAbstReach requires a set of transition predicates in order to compute an over-approximation of the transition invariant. Finding the right set of transition predicates that yields a sufficiently precise over-approximation is a difficult task.

Analysis of counterexample lassos We present a notion of lasso-shaped counterexample that is suitable for refining transition predicate abstraction. Such counterexamples consist of a stem and a loop. The step part represents a sequence of program transitions that leads to a loop in the program, while the loop part is a sequence of program transitions that represents a possible execution through such a loop.

First, we illustrate counterexample lassos using an example.

Example 22. We consider the transition invariant computed in Example 16 by applying TransAbstReach. We assume that $ReachStates^{\#} = \{true\}$

```
    function MakeLasso
    input
      $\ddot{\psi}$ - reachable abstract transition
      $Parent$ - predecessor relation for abstract states
      $TransParent$ - predecessor relation for abstract transitions
    begin
1     $loop$ := empty sequence
2     $\ddot{\varphi}'$ := $\ddot{\psi}$
3     while exist $\ddot{\varphi}$ and $\rho$ such that $(\ddot{\varphi}, \rho, \ddot{\varphi}') \in TransParent$ do
4       $loop$ := $\rho$ . $loop$
5       $\ddot{\varphi}'$ := $\ddot{\varphi}$
6     $(\varphi \wedge unchanged(V))$ := $\ddot{\varphi}$
7     $stem$ := MakePath($\varphi$, $Parent$)
8     return ($stem$, $loop$)
    end
```

Fig. 1.13 Lasso computation.

used for this computation was obtained by applying AbstReach and that $Parent = \{(true, \rho_1, true)\}$ was obtained as a result.

We observe that $ReachTrans^{\#}$ is not disjunctively well-founded, since it contains $goto(\ell_2, \ell_2)$ that is not well-founded. Similarly to the treatment of counterexamples in predicate abstraction-based invariant computation, we use *TransParent* to determine a sequence of program transitions that led to the computation of $goto(\ell_2, \ell_2)$, which we call loop. We observe that $(true \wedge unchanged(x, y, z), \rho_1, goto(\ell_1, \ell_2)) \in TransParent$, hence the last element of the loop is the transition ρ_2. Furthermore, since $true \wedge unchanged(x, y, z)$ does not appear in the third position of some element of *TransParent*, we conclude that no more elements appear in the loop. Now we determine the stem part by applying MakePath on *true*, which is obtained from $true \wedge unchanged(x, y, z)$ by omitting equalities $unchanged(x, y, z)$, and *Parent*. The result is a stem that consists only of ρ_1, which finishes the computation of the counterexample lasso. $\square$

The algorithm MakeLasso shown in Figure 1.13 implements the lasso construction as described in the above example. MakeLasso proceeds similarly to MakePath and calls it as a sub-routine in line 7 after the loop part is constructed. We detect that the loop construction is finished when there is no predecessor according to *TransParent*. The stem part is constructed using the information collected during abstract reachability computation and provided as *Parent*. The starting point for the stem computation is obtained

```
    function FeasibleLasso
    input
      $\rho_1 \dots \rho_n$ - stem transitions
      $\rho'_1 \dots \rho'_m$ - loop transitions
    begin
1     $\varphi \ := \ post(\varphi_{init}, \rho_1 \circ \dots \circ \rho_n)$
2     $\ddot{\varphi} \ := \ (\varphi \wedge unchanged(V)) \circ \rho'_1 \circ \dots \circ \rho'_m$
3     if $\neg$well-founded($\ddot{\varphi}$) then
4        return true
5     else
6        return false
    end
```

Fig. 1.14 Feasibility of a lasso.

using pattern matching in line 6. Here, we exploit how *Worklist* is initialized by TransAbstReach in line 4, see Figure 1.5.

Once the lasso counterexample is constructed, we analyse whether the transition predicate abstraction can be refined in order to rule out the discovered counterexample. First, we illustrate this step by using an example.

Example 23. We consider the lasso computed in Example 22 that consists of the stem ρ_1 and the loop ρ_2. We compute the set of states φ that are reachable by applying the stem and obtain

$$\varphi = post(\varphi_{init}, \rho_1) = (at\_\ell_1 \wedge y \geq z) \ .$$

We use this set of states when initializing the relational composition of program transitions along the loop part with

$$at\_\ell_2 \wedge y \geq z \wedge unchanged(x, y, z) \ .$$

The result of the composition – this time without applying transition predicate abstraction – is

$$\begin{aligned} \ddot{\varphi} &= (at\_\ell_2 \wedge y \geq z \wedge unchanged(x, y, z)) \circ \rho_2 \\ &= (goto(\ell_2, \ell_2) \wedge y \geq z \wedge x + 1 \leq y \wedge x' = x + 1 \wedge unchanged(y, z)) \ . \end{aligned}$$

The obtained relation $\ddot{\varphi}$ is well-founded, which can be easily checked since it is represented by a simple program loop without further nesting or branching statements. A ranking function that witnesses the termination of $\ddot{\varphi}$ is $y - x$. Every time the relation is applied, the value of $y - x$ decreases. Furthermore,

$\ddot{\varphi}$ can be applied only if $y - x \geq 0$. We conclude that the discovered counterexample lasso is spurious, i.e., there is no infinite program computation that follow the stem and then repeats the loop part forever. □

The algorithm FEASIBLELASSO shown in Figure 1.14 automates the steps executed in the above example. FEASIBLELASSO takes as input a lasso obtained by applying MAKELASSO and performs a check in the given lasso can yield an infinite computation. The implementation of the predicate *well-founded* is out of scope of this chapter. There exist efficient algorithms for this task that exploit the lasso shape, e.g., [51].

Finally, we show how transition predicates can be discovered from a spurious counterexample lasso.

Example 24. We consider the feasibility check presented in Example 23. We observe that the following implications were established.

$$post(\varphi_{init}, \rho_1) \models (at\_\ell_1 \wedge y \geq z)$$
$$(at\_\ell_2 \wedge y \geq z \wedge unchanged(x, y, z)) \circ \rho_2 \models y - x \geq 0 \wedge y' - x' \leq y - x - 1$$

Hence, to eliminate the spurious counterexample we can use the assertions *true* and $y - x \geq 0 \wedge y' - x' \leq y - x - 1$. When $y - x \geq 0$ and $y' - x' \leq y - x - 1$ are included in the set of transition predicates *TransPreds* used by the abstraction function $\ddot{\alpha}$ the algorithm TRANSABSTREACH will not discover the spurious lasso consisting of ρ_1 and *relLoop* again. Example 14 shows the outcome of applying TRANSABSTREACH when using a refined set of transition predicates *TransPreds*. □

We present an algorithm REFINELASSO in Figure 1.15 that discovers transition predicates from a spurious counterexample lasso. The algorithm is presented in a declarative way and we omit details of a particular implementation of line 1. There exist efficient implementations for this task that rely on similar techniques as presented in Section 1.7, e.g., [32].

Algorithm for counterexample guided transition predicate abstraction refinement We put together the algorithms for the construction and analysis of lasso shaped counterexamples presented above together with the algorithm for transition predicate abstraction. The resulting algorithm TRANSABSTREFINELOOP can find a disjunctively well-founded transition invariant automatically by automatically discovering an adequate set of transition predicates. See Figure 1.16.

TRANSABSTREFINELOOP proceeds in similar steps as ABSTREFINELOOP presented in Section 1.6. In fact, we use ABSTREFINELOOP to compute an over-approximation of reachable program states. We start with the empty set of predicates and transition predicates and extend them every time a

```
function RefineLasso
input
  $\rho_1 \ldots \rho_n$ - stem transitions
  $\rho'_1 \ldots \rho'_m$ - loop transitions
begin
1   $\varphi_0, \ldots, \varphi_n, \ddot{\varphi}_1, \ldots, \ddot{\varphi}_m$ := compute such that
2     $(\varphi_{init} \models \varphi_0) \wedge$
3     $(post(\varphi_0, \rho_1) \models \varphi_1) \wedge \ldots \wedge (post(\varphi_{n-1}, \rho_n) \models \varphi_n) \wedge$
4     $((\varphi_n \wedge unchanged(V)) \circ \rho'_1 \models \ddot{\varphi}_1) \wedge \ldots \wedge (\ddot{\varphi}_{m-1} \circ \rho'_m \models \ddot{\varphi}_m) \wedge$
5     $well\text{-}founded(\ddot{\varphi}_m)$
6   return $(\{\varphi_0, \ldots, \varphi_n\}, \{\ddot{\varphi}_1, \ldots, \ddot{\varphi}_m\})$
end
```

Fig. 1.15 Abstraction refinement guided by a lasso.

```
function TransAbstRefineLoop
begin
1   $Preds$ := $\emptyset$
2   $TransPreds$ := $\emptyset$
3   repeat
4     $(ReachStates^{\#}, Parent)$ := AbstReach($Preds$)
5     $(ReachTrans^{\#}, TransParent)$ := TransAbstReach($TransPreds$)
6     if exists $\ddot{\psi} \in ReachTrans^{\#}$ such that $\neg well\text{-}founded(\ddot{\psi})$ then
7       $(stem, loop)$ := MakeLasso($\ddot{\psi}, Parent, TransParent$)
8       if FeasibleLasso($stem, loop$) then
9         return "counterexample lasso: $stem, loop$ "
10      else
11        $(NewPreds, NewTransPreds)$ := RefineLasso($stem, loop$)
12        $Preds$ := $NewPreds \cup Preds$
13        $TransPreds$ := $NewTransPreds \cup TransPreds$
14    else
15      return "program terminates"
end.
```

Fig. 1.16 Transition predicate abstraction and refinement loop.

counterexample lasso is discovered. The counterexample discovery takes place during the computation of a transition invariant using TransAbstReach. If a counterexample lasso is found, its stem part is used to refine the set of predicates *Preds*. The set of additional transition predicates is determined by considering both the stem and the loop parts.

Similarly to abstraction refinement for safey, we observe that TransAbstRefineLoop never analyzes the same counterexample twice, i.e., the abstraction refinement process using RefineLasso makes progress at each iteration.

1.7 Solving refinement constraints for predicate abstraction

The algorithm RefinePath in Figure 1.9 takes as input an infeasible sequence of program transitions $\rho_1 \dots \rho_n$ and computes sets of states $\varphi_0, \dots, \varphi_n$ satisfying the following conditions.

$$\begin{array}{l} \varphi_{init} \models \varphi_0 \\ post(\varphi_0, \rho_1) \models \varphi_1 \\ \dots \\ post(\varphi_{n-1}, \rho_n) \models \varphi_n \\ \varphi_n \wedge \varphi_{err} \models false \ . \end{array}$$

Since $\rho_1 \dots \rho_n$ is infeasible, the above conditions are satisfiable. In general, several solutions may exist. We describe how the least, the greatest, and an intermediate solution can be computed.

1.7.1 Least solution

We obtain the least solution by applying the post-condition function in the following way.

$$\begin{array}{rcl} \varphi_0 &=& \varphi_{init} \\ \varphi_1 &=& post(\varphi_0, \rho_1) \\ \dots \\ \varphi_n &=& post(\varphi_{n-1}, \rho_n) \end{array} \quad (1.22)$$

Note that since the least solution ensures that for each $1 \leq i \leq n$ we have

$$\varphi_i = post(\varphi_{init}, \rho_1 \circ \dots \circ \rho_i) \ ,$$

and guarantee that $\varphi_n \wedge \varphi_{err} \models false$.

Sometimes the least solution is not useful for refining the abstraction, since the resulting abstraction is too precise. As a result, the iteration in AbstRefineLoop may not terminate as the abstract reachability computation is almost equivalent to the reachability computation without abstraction.

Example 25. We illustrate how a least solution is computed using an example program shown in Figure 1.1.

Let $\rho_1\rho_3\rho_5$ be a counterexample path discovered by AbstRefineLoop. For this path, we obtain the following least solution of the constraints defined by RefinePath.

$$\begin{aligned}
\varphi_0 &= \varphi_{init} &&= at\_\ell_1 \\
\varphi_1 &= post(\varphi_0, \rho_1) &&= (at\_\ell_2 \land y \geq z) \\
\varphi_2 &= post(\varphi_1, \rho_3) &&= (at\_\ell_3 \land y \geq z \land x \geq y) \\
\varphi_3 &= post(\varphi_2, \rho_5) &&= false
\end{aligned}$$

The obtained refinement will ensure that the path $\rho_2\rho_3\rho_5$ will not be considered a counterexample during subsequent iterations of the refinement loop in AbstRefineLoop. □

1.7.2 Greatest solution

First, we define an auxiliary *weakest pre-condition* function *wp* as follows. Let φ be a formula over V and let ρ be a formula over V and V'. Then, we define:

$$wp(\varphi, \rho) \ = \ \forall V' : \rho \rightarrow \varphi[V'/V]. \tag{1.23}$$

For example, a transition ρ_2 from Figure 1.1 results in the following weakest precondition.

$$\begin{aligned}
& wp(at\_\ell_2 \land x \geq z, \rho_2) \\
& \quad = \forall V' : pc = \ell_2 \land x + 1 \leq y \land x' = x + 1 \land y' = y \land z' = z \land pc' = \ell_2 \\
& \qquad\qquad \rightarrow pc' = \ell_2 \land x' \geq z \\
& \quad = \neg(\exists V' : pc = \ell_2 \land x + 1 \leq y \land x' = x + 1 \land y' = y \land z' = z \land pc' = \ell_2 \land \\
& \qquad\qquad\quad \neg(pc' = \ell_2 \land x' \geq z)) \\
& \quad = \neg(\exists V' : pc = \ell_2 \land x + 1 \leq y \land \neg(\ell_2 = \ell_2 \land x + 1 \geq z)) \\
& \quad = (pc = \ell_2 \land x + 1 \leq y \rightarrow \ell_2 = \ell_2 \land x + 1 \geq z) \\
& \quad = (at\_\ell_2 \land x + 1 \leq y \rightarrow x + 1 \geq z)
\end{aligned}$$

We obtain the greatest solution of the refinement constraints for a given counterexample path as follows.

$$\begin{aligned} \varphi_n &= \neg\varphi_{err} \\ \varphi_{n-1} &= wp(\varphi_n, \rho_n) \\ \dots & \\ \varphi_0 &= wp(\varphi_1, \rho_1) \end{aligned} \tag{1.24}$$

That is, the greatest solution is computed incrementally by traversing the counterexample path backwards.

Similarly to the least solution, sometimes the greatest solution is not useful for refining the abstraction, since the resulting abstraction is too coarse. As a result, the iteration in AbstRefineLoop may not terminate as the abstract reachability computation is almost equivalent to the backward reachability computation without abstraction that expands the set of states definitely leading to an error state.

Example 26. We illustrate how a greatest solution is computed using an example program shown in Figure 1.1.

Let $\rho_1\rho_3\rho_5$ be a counterexample path discovered by AbstRefineLoop. For this path, we obtain the following greatest solution of the constraints in RefinePath.

$$\begin{aligned} \varphi_3 &= \neg\varphi_{err} &&= \neg at\_\ell_5 \\ \varphi_2 &= wp(\varphi_3, \rho_5) &&= (at\_\ell_3 \to x \geq z) \\ \varphi_1 &= wp(\varphi_2, \rho_3) &&= (at\_\ell_2 \wedge x \geq y \to x \geq z) \\ \varphi_0 &= wp(\varphi_1, \rho_1) &&= true \end{aligned}$$

Again, the obtained refinement will result in the discovery of the counterexample path $\rho_2\rho_3\rho_5$ during the next iteration in AbstRefineLoop, as witnessed by the following validities.

$$\begin{aligned} &\varphi_{init} \models \varphi_0 \\ &post(\varphi_0, \rho_1) = (at\_\ell_2 \wedge y \geq z) \models \varphi_1 \\ &post(\varphi_1, \rho_3) = (at\_\ell_3 \wedge x \geq y \wedge x \geq z) \models \varphi_2 \\ &post(\varphi_2, \rho_5) = false \models \varphi_3 \end{aligned}$$

We observe that the reachability computation using refined abstraction does not reach any error states along the path $\rho_1\rho_3\rho_5$. □

1.7.3 Intermediate solution using interpolation

We illustrate how an intermediate solution can be computed by a technique called interpolation [25,38]. Interpolation takes as input two mutually unsatisfiable formulas φ_1 and φ_2, i.e., $\varphi_1 \wedge \varphi_2 \models \mathit{false}$, and returns a formula φ such that i) φ is expressed over common symbols of φ_1 and φ_2, ii) $\varphi_1 \models \varphi$, and iii) $\varphi \wedge \varphi_2 \models \mathit{false}$. Let *inter* be an interpolation function such that $\mathit{inter}(\varphi_1, \varphi_2)$ is an interpolant for φ_1 and φ_2.

The following sequence of interpolation computations can be used to find a solution for constraints defined by RefinePath.

$$\begin{aligned}
&\varphi_0 = \mathit{inter}(\varphi_{init}, (\rho_1 \circ \ldots \circ \rho_n) \wedge \varphi_{err}[V'/V]) \\
&\varphi_1 = \mathit{inter}(\mathit{post}(\varphi_0, \rho_1), (\rho_2 \circ \ldots \circ \rho_n) \wedge \varphi_{err}[V'/V]) \\
&\ldots \\
&\varphi_{n-1} = \mathit{inter}(\mathit{post}(\varphi_{n-2}, \rho_{n-1}), \rho_n \wedge \varphi_{err}[V'/V]) \\
&\varphi_n = \mathit{inter}(\mathit{post}(\varphi_{n-1}, \rho_n), \varphi_{err}[V'/V])
\end{aligned} \tag{1.25}$$

Intermediate solutions can avoid the deficiencies of least and greatest solutions described above, although they still do not guarantee convergence of the abstraction refinement loop.

Example 27. We illustrate how an intermediate solution is computed using an example program shown in Figure 1.1.

Let $\rho_1\rho_3\rho_5$ be a counterexample path discovered by AbstRefineLoop. For this path, we obtain the following intermediate solution of the constraints in RefinePath.

$$\begin{aligned}
\varphi_0 &= \mathit{inter}(\varphi_{init}, (\rho_1 \circ \rho_3 \circ \rho_5) \wedge \varphi_{err}[V'/V]) &&= \mathit{true} \\
\varphi_1 &= \mathit{inter}(\mathit{post}(\varphi_0, \rho_1), (\rho_3 \circ \rho_5) \wedge \varphi_{err}[V'/V]) &&= y \geq z \\
\varphi_2 &= \mathit{inter}(\mathit{post}(\varphi_1, \rho_3), \rho_5 \wedge \varphi_{err}[V'/V]) &&= x \geq z \\
\varphi_3 &= \mathit{inter}(\mathit{post}(\varphi_2, \rho_5), \varphi_{err}[V'/V]) &&= \mathit{false}
\end{aligned}$$

The following validities show that $\rho_1\rho_3\rho_5$ will not be considered a counterexample during subsequent refinement iterations.

$$\begin{aligned}
&\varphi_{init} \models \varphi_0 \\
&\mathit{post}(\varphi_0, \rho_1) = (\mathit{at\_}\ell_2 \wedge y \geq z) \models \varphi_1 \\
&\mathit{post}(\varphi_1, \rho_3) = (\mathit{at\_}\ell_3 \wedge x \geq y \wedge y \geq z) \models \varphi_2 \\
&\mathit{post}(\varphi_2, \rho_5) = \mathit{false} \models \varphi_3
\end{aligned}$$

□

1.8 Tools

We have presented the base algorithm for predicate abstraction and transition predicate abstraction. Practical tools introduce a variety of optimizations of the base algorithm.

Predicate abstraction SLAM [4], BLAST [38, 39], Magic [14], Murphi [26], and SatAbs [17] implement different levels of precision, ranging from Cartesian and full boolean predicate abstraction [3]. CPAChecker [9], F-Soft [41], and UFO [1] integrate predicate abstraction with data flow analysis and abstract interpretation. Synergy [33] and Yogi [50] integrate predicate abstraction with underapproximation based on dynamic execution. ARMC [54] implements Cartesian predicate abstraction and uses constraint based interpolation to discover predicates. SLAB [28] implements the refinement of an abstract transition system in a top-down way. Impact [49] and Wolverine [45] resort to a particular form of predicate abstraction where each refinement steps adds a single predicate. Ultimate Automizer [36] uses predicates to construct a proof in the form of a finite automaton that approximates the language of program traces.

Arrays and heaps BLAST [43], Indexed Predicate Abstraction [46], and universally quantified Horn solver [11] compute universally quantified array invariants with predicate abstraction in order to deal with the ranges of array indices and properties of values stored in arrays. Bohne [56, 57] verifies complex data structures that are implemented on the heap (modeled as a graph) by inferring node predicates in the style of TVLA [59].

Beyong procedural programs HSF [32] relies on predicate abstraction to solve recursive Horn constraints, which serves as a backend solver for proving termporal properties of programs with procedures, multi-threaded programs and higher order funcitonal programs. Threader [34, 35] relies on predicate abstraction to compute rely/guarantee and Owicki/Gries proofs for multi-threaded programs. Liquid Types [58] uses a form of predicate abstraction in the style of Houdini [29] to infer refinement types for proving safety of higher order functional programs.

Beyond safety ARMC [54] uses transition predicate abstraction as described in [53] to prove termination and other liveness properties. Terminator [19, 20] reduces transition predicate abstraction to predicate abstraction via a syntactic transformation of the program in order to prove termination of systems code. T2 [12,22] computes transition invariants using techniques as in Impact [49]. LoopFrog computes transition invariants to analyze termination of programs at a bitlevel semantics. LTA [48] uses algorithmic learning-based techniques for the generation of transition predicates. CTA [44] computes 'compositional' transition invariants. Ultimate Automizer [36] uses transition predicates to construct a proof in the form of a finite Bü chi automaton that approximates the language of infinite program traces. Several tools including AProVe [13] and ACL2 [15] use the size-change principle [47], whose formal connection to transition predicate abstraction (without refinement) is studied in [37]. An explanation why transition predicate abstraction works for termination analysis is given in the abstract interpretation framework in [24]. HSF [32] relies on transition predicate abstraction in combination with abstract inference to find well-founded models for Horn constraints.

Beyond verification Existentially quantified Horn solver [7] uses predicate abstraction to discover witness existential quantification in Horn constraints and to synthesize winning strategies for LTL games [6].

1.9 Conclusion and discussion

We presented an over-approximation technique called predicate abstraction and showed how it can be applied for proving reachability and termination properties of programs. We automate predicate abstraction by relying on a decision procedure for computing entailments. An adequate set of predicates can be discovered automatically by exploring spurious counterexamples.

Our presentation aimed at basic principles of predicate abstraction and left uncovered the many variations of predicate abstraction studied in the literature and implemented in tools. Chapter 16 shows how the process of computing over-approximating transition relations using predicates can be decoupled from the fixpoint computation. Chapter 18 shows how predicate abstraction can be combined with data flow analysis, thus only requiring decision procedure calls for intricate reasoning that is difficult to support in classical data flow domains.

The presented approach to combine predicates when computing an over-approximation only considers conjunction. This approach is called Cartesian abstraction in the literature, since each predicate is treated in isolation. Alternatively, over-approximation using Boolean combinations of predicates can be used, at a higher cost of computing the abstraction function. We refer to [42] for a survey of this and related techniques and variations. In this regards, we point out that so-called large block encoding techniques that operate on compound program transitions with rich Boolean structure can be effectively leverage the advances in state-of-the-art decision procedures and thus can offer both precision and efficiency [8, 10] .

References

1. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *CAV*, pages 672–678, 2012.

2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Languages Design and Implementation*, pages 203–213. ACM, 2001.
3. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking c programs. *STTT*, 5(1):49–58, 2003.
4. T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.
5. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, Lecture Notes in Computer Science 1885, pages 113–130. Springer-Verlag, 2000.
6. T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, pages 221–234, 2014.
7. T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified Horn clauses. In *CAV*, pages 869–882, 2013.
8. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32. IEEE, 2009.
9. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.
10. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *FMCAD*, pages 189–197. FMCAD, 2010.
11. N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *SAS*, pages 105–125, 2013.
12. M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 413–429. Springer, 2013.
13. M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2012.
14. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE 03: Software Engineering*, pages 385–395. IEEE, 2003.
15. H. R. Chamarthi, P. C. Dillinger, P. Manolios, and D. Vroon. The ACL2 Sedan theorem proving system. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2011.
16. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, Lecture Notes in Computer Science 1855, pages 154–169. Springer-Verlag, 2000.
17. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, pages 570–574, 2005.
18. E. M. Clarke, R. P. Kurshan, and H. Veith. The localization reduction and counterexample-guided abstraction refinement. In Z. Manna and D. Peled, editors, *Essays in Memory of Amir Pnueli*, volume 6200 of *Lecture Notes in Computer Science*, pages 61–71. Springer, 2010.
19. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI 06: Programming Languages Design and Implementation*, pages 415–426. ACM, 2006.
20. B. Cook, A. Podelski, and A. Rybalchenko. Terminator: beyond safety. In *CAV*, pages 415–418, 2006.
21. B. Cook, A. Podelski, and A. Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3):369–387, 2009.
22. B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In N. Piterman and S. A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2013.

23. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *POPL*, pages 238–252. ACM, 1977.
24. P. Cousot and R. Cousot. An abstract interpretation framework for termination. In J. Field and M. Hicks, editors, *POPL*, pages 245–258. ACM, 2012.
25. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen Theorem. *J. Symb. Log.*, 22(3):250–268, 1957.
26. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV 99: Computer-Aided Verification*, Lecture Notes in Computer Science 1633, pages 160–171. Springer-Verlag, 1999.
27. R. Dimitrova and A. Podelski. Is lazy abstraction a decision procedure for broadcast protocols? In *VMCAI 08: Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science 4905, pages 98–111. Springer-Verlag, 2008.
28. K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In *TACAS*, pages 271–274, 2010.
29. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.
30. R. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.
31. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-Aided Verification*, Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, 1997.
32. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
33. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127, 2006.
34. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344, 2011.
35. A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV*, pages 412–417, 2011.
36. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, pages 471–482, 2010.
37. M. Heizmann, N. D. Jones, and A. Podelski. Size-change termination and transition invariants. In *Static Analysis*, pages 22–50. Springer, 2011.
38. T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.
39. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.
40. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
41. F. Ivančić, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-SOFT. In *ICCD*, pages 297–308, 2005.
42. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, Oct. 2009.
43. R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV 07: Computer-Aided Verification*, pages 193–206, 2007.
44. D. Kroening, N. Sharygina, A. Tsitovich, and C. Wintersteiger. Termination analysis with compositional transition invariants. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 89–103. Springer Berlin Heidelberg, 2010.
45. D. Kroening and G. Weissenbacher. Interpolation-based software verification with Wolverine. In *CAV*, pages 573–578, 2011.

46. S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.
47. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In C. Hankin and D. Schmidt, editors, *POPL*, pages 81–92. ACM, 2001.
48. W. Lee, B.-Y. Wang, and K. Yi. Termination analysis with algorithmic learning. In *Computer Aided Verification*, pages 88–104. Springer, 2012.
49. K. L. McMillan. Lazy abstraction with interpolants. In *CAV 2006*, Lecture Notes in Computer Science, pages 123–136. Springer-Verlag, 2006.
50. A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The Yogi project: Software property checking via static analysis and testing. In *TACAS*, pages 178–181, 2009.
51. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.
52. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS 04: Logic in Computer Science*, pages 32–41. IEEE, 2004.
53. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL 05: Principles of Programming Languages*, pages 132–144. ACM, 2005.
54. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL 07: Practical Aspects of Declarative Programming*, Lecture Notes in Computer Science 4354, pages 245–259. Springer-Verlag, 2007.
55. A. Podelski and A. Rybalchenko. Transition invariants and transition predicate abstraction for program termination. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–10. Springer, 2011.
56. A. Podelski and T. Wies. Boolean heaps. In C. Hankin and I. Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2005.
57. A. Podelski and T. Wies. Counterexample-guided focus. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 249–260, New York, NY, USA, 2010. ACM.
58. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 158–169, 2008.
59. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.