## 0.1 Introduction

As the title of this chapter announces, we are interested in a specific category of algorithms. The input is a program. The output is an answer to the question whether the program is correct. Correctness is expressed by one of two properties: non-reachability of a distinguished state, or termination. As the term *software* in the title indicates, we consider classes of programs for which both, non-reachability and termination, are not decidable. We distinguish between algorithms that, for some inputs, do not terminate and algorithms that always terminate but, for some inputs, the output is a *don't know* answer.

The specificity of our category of algorithms lies in the way the algorithms call *decision procedures* as subroutines. Decision procedures in the sense used in this chapter are algorithms to decide whether the inclusion between two given sets of states holds. The two sets of states are represented by a logical formula. The inclusion between the sets reduces to the validity of a formula in a logical theory. The logical theory corresponds to kind of data in the considered class of programs. In this chapter, we use decision procedures as an oracle. We abstract away from the fact that the logical theory may be undecidable. The behaviour of our algorithms is left unspecified if the oracle returns a *don't know* answer (or no answer at all).

A set of states is sometimes called a predicate; a superset of a set is sometimes called its abstraction; the terminology *predicate abstraction* refers to the fact that the sets used for abstraction are formed by Boolean combination from a finite number of given sets; it is a convention to refer to only that finite number of given sets as *predicates* (thus reserving the terminology for a special case). The more predicates, the more sets are available for the abstraction of a set by a superset. In this context, *abstraction refinement* is simply the process of adding new predicates. The crux of the algorithms is the (*counterexample-guided*) way in which new predicates are constructed.

Software verification with predicate abstraction is an ongoing research topic (see our list of references). 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 re-appearing 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.

## 0.2 Preliminaries

In this section, we describe programs, computations, and properties. To see an example of a program early, go to Figure 0.1.

### *0.2.1 Programs*

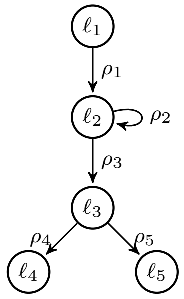
A *program* $Prog = (V, pc, \varphi_{init}, \mathcal{R}, \varphi_{err})$ consists of

- $V$ - a finite tuple of *program variables*,
- $pc$ - a *program counter variable* that is included in $V$,
- $\varphi_{init}$ - an *initiation condition* given by a formula over $V$,
- $\mathcal{R}$ - a finite set of ("single-statement") *transition relations*, where each transition relation $\rho \in \mathcal{R}$ is given by a formula over $V$ and their primed versions $V'$,
- $\varphi_{err}$ - an *error condition* given by a formula over $V$.

Each program variable is assigned a *domain* of values. A *program state* is a function that assigns each program variable a value from its respective domain. Let $\Sigma$ be the set of program states. A formula with free variables in $V$ represents a set of program states. A formula with free variables in $V$ and $V'$ represents

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

(a)



(b)

$$\begin{aligned}
\rho_1 &= (\mathit{move}(\ell_1, \ell_2) \wedge y \geq z \wedge \mathit{skip}(x, y, z))\\
\rho_2 &= (\mathit{move}(\ell_2, \ell_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge \mathit{skip}(y, z))\\
\rho_3 &= (\mathit{move}(\ell_2, \ell_3) \wedge x \geq y \wedge \mathit{skip}(x, y, z))\\
\rho_4 &= (\mathit{move}(\ell_3, \ell_4) \wedge x \geq z \wedge \mathit{skip}(x, y, z))\\
\rho_5 &= (\mathit{move}(\ell_3, \ell_5) \wedge x + 1 \leq z \wedge \mathit{skip}(x, y, z))
\end{aligned}$$

(c)

**Fig. 0.1** An example program (a), its control-flow graph (b), and its transition relations (c). Formally, the program is given by $\mathit{Prog} = (V, pc, \varphi_{init}, \mathcal{R}, \varphi_{err})$ where $V = (pc, x, y, z)$ is the tuple of program variables, $pc$ is the program counter variable, $\mathcal{R} = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$ is the set of ("single-statement") 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 *move* and *skip* as an abbreviation, for example $\mathit{move}(\ell_1, \ell_2)$ stands for $(pc = \ell_1 \wedge pc' = \ell_2)$ and $\mathit{skip}(x, y, z)$ represents $(x' = x \wedge y' = y \wedge z' = z)$.

a binary relation over program states, where the first component of each pair assigns values to $V$ and the second component of the pair assigns values to $V'$. We identify formulas with sets and relations that they represent. Accordingly, we identify the logical consequence relation between formulas $\models$ with the set inclusion $\subseteq$. Furthermore, we identify the satisfaction relation between valuations and formulas, which is denoted by $\models$, with the membership relation $\in$.

*Example 1.* For example, we consider the program shown in Figure 0.1 that has program variables $V = (pc, x, y, z)$. The program variables $x$, $y$, and $z$ range over integers. The set of control locations is $\mathcal{L} = \{\ell_1, \dots \ell_5\}$. A formula $y \geq z$ represents the set of program states in which the value of the variable $y$ is greater than the value of $z$. Let $s$ be a program state that assigns 1, 3, 2, and $\ell_1$ to the program variables $x$, $y$, $z$, and $pc$, respectively. Then, we have $s \models y \geq z$. Furthermore, we have $y \geq z \models y + 1 \geq z$. □

Each state that satisfies the initiation condition $\varphi_{init}$ is called an *initial* state. Each state that satisfies the error condition $\varphi_{err}$ is called an *error* state. The program transition relation $\rho_{\mathcal{R}}$ is the union of the "single-statement" transition relations, i.e.,

$$\rho_{\mathcal{R}} = \bigvee_{\rho \in \mathcal{R}} \rho \; . \tag{0.1}$$

A pair of states $(s, s')$ is connected by a program transition if it lies in the program transition relation $\rho_{\mathcal{R}}$, i.e., if $(s, s') \models \rho_{\mathcal{R}}$.

Let $\mathcal{L}$ be the domain of the program counter variable $pc$, i.e., the set of control locations of the program. To simplify the notation in the examples, we introduce the following abbreviations, where $\ell \in \mathcal{L}$ is a control location and $v_1, \dots, v_n$ are program variables.

$$\begin{aligned} at\_\ell &= (pc = \ell) \\ at'\_\ell &= (pc' = \ell) \\ move(\ell, \ell') &= (at\_\ell \wedge at'\_\ell) \\ skip(v_1, \ldots, v_n) &= (v_1' = v_1 \wedge \ldots \wedge v_n' = v_n) \end{aligned} \tag{0.2}$$

*Example 2.* Our example program has an initiation condition $\varphi_{init} = (pc = at\_\ell_1)$ and an error condition $\varphi_{err} = (pc = at\_\ell_5)$. Program transitions $\mathcal{R} = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$ form a control-flow graph as shown in Figure 0.1(b). The corresponding transition relations are in Figure 0.1(c). Here, the first transition relation $\rho_1$ requires that the value of program counter is equal to $\ell_1$ and that $y \geq z$ for the transition to be applicable. After executing the transition, the program counter value changes to $\ell_2$ and the values of $x$, $y$, and $z$ are not modified. The transition relation of the program consists of the disjunction $\rho_{\mathcal{R}} = \rho_1 \vee \rho_2 \vee \rho_3 \vee \rho_4 \vee \rho_5$. □

A *program computation* is either a finite or an infinite sequence of program states $s_1, s_2, \ldots$ that satisfies the following three conditions.

- The first element is an initial state, i.e., $s_1 \models \varphi_{init}$.
- Each pair of consecutive states $(s_i, s_{i+1})$ is connected by a program transition, i.e., $(s_i, s_{i+1}) \models \rho_{\mathcal{R}}$.
- If the sequence is finite then the last element does not have any successors wrt. the program transition relation $\rho_{\mathcal{R}}$, i.e., if the last element is $s_n$, there is no state $s$ such that $(s_n, s) \models \rho_{\mathcal{R}}$.

*Example 3.* In the example program, we consider a program computation connected by following the sequence of transitions $\rho_1, \rho_2, \rho_2, \rho_3, \rho_4$. We represent states as tuples of values of the program variables $pc$, $x$, $y$, and $z$, respectively.

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

The last program state does not any successors wrt. the program transition relation. □

### 0.2.2 Correctness

We consider only two properties of program computations. These properties are concerned with the reachability of particular program states and the finiteness, i.e., termination, of the computation. Checking an expressive class of temporal properties can be reduced to reasoning about reachability and termination.

*Safety* A state is *reachable* if it occurs in a program computation. A program is *safe* if no error state is reachable.

Let $\varphi_{reach}$ denote the set of reachable program states. A program is *safe* if and only if no error state lies in $\varphi_{reach}$, i.e.,

$$\varphi_{err} \wedge \varphi_{reach} \models \mathit{false}\ . \tag{0.3}$$

*Example 4.* In our example program, the state $(\ell_3, 3, 3, 2)$ is reachable, as witnessed by the above computation. The set of reachable program states is

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

Our program is safe, since $\varphi_{reach}$ does not contain any states at the control location $\ell_5$. □

*Termination* A program *terminates* if every computation is finite. A binary relation is *well-founded* if it does not admit any infinite chains. The restriction of the program transition relation $\rho_{\mathcal{R}}$ to the reachable program states is given by $\rho_{\mathcal{R}} \wedge \varphi_{reach}$ (the conjunction of a formula over $V$ and $V'$ and a formula over $V$). A program terminates if and only if the binary relation $\rho_{\mathcal{R}} \wedge \varphi_{reach}$ is well-founded.

*Example 5.* For our example, we obtain the following restriction of the program transition relation to reachable states.

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

The restriction consists of four disjuncts, since the transition relation $\rho_5$ does not intersect with $\varphi_{reach}$. 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. □

### *0.2.3 Reachability and transitive closure computation*

Let $\varphi$ be a formula over $V$ and let $\rho$ be a formula over $V$ and $V'$. We define a *post-condition* function *post* as follows.

$$\mathit{post}(\varphi, \rho) \;=\; \exists V'' : \varphi[V''/V] \wedge \rho[V''/V][V/V'] \tag{0.4}$$

An application $\mathit{post}(\varphi, \rho)$ computes the image of the set $\varphi$ under the relation $\rho$. Furthermore, for a natural number $n$ we define $\mathit{post}^n(\varphi, \rho)$ as follows.

$$\mathit{post}^n(\varphi, \rho) \;=\; \begin{cases}\varphi & \text{if } n = 0\\ \mathit{post}(\mathit{post}^{n-1}(\varphi, \rho), \rho) & \text{otherwise}\end{cases} \tag{0.5}$$

By $\mathit{post}^n(\varphi, \rho)$ we represent the $n$-fold application of the *post* function to $\varphi$ with respect to $\rho$. We observe the following useful property of the post-condition function.

$$\begin{aligned}&\forall \varphi\, \forall \rho_1\, \forall \rho_2 : \mathit{post}(\varphi, \rho_1 \vee \rho_2) = (\mathit{post}(\varphi, \rho_1) \vee \mathit{post}(\varphi, \rho_2))\\ &\forall \varphi_1\, \forall \varphi_2\, \forall \rho : \mathit{post}(\varphi_1 \vee \varphi_2, \rho) = (\mathit{post}(\varphi_1, \rho) \vee \mathit{post}(\varphi_2, \rho))\end{aligned} \tag{0.6}$$

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

*Example 6.* For example, given the transition relation $\rho_2$ and the program variables $V = (pc, x, y, z)$ from our example program, we compute the following post condition.

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

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

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

□

We characterize $\varphi_{reach}$ using *post* as follows.

$$\begin{aligned}
\varphi_{reach} &= \varphi_{init} \vee \mathit{post}(\varphi_{init}, \rho_{\mathcal{R}}) \vee \mathit{post}(\mathit{post}(\varphi_{init}, \rho_{\mathcal{R}}), \rho_{\mathcal{R}}) \vee \dots \\
&= \textstyle\bigvee_{i \geq 0} \mathit{post}^i(\varphi_{init}, \rho_{\mathcal{R}})
\end{aligned} \tag{0.7}$$

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

*Example 7.* We compute $\varphi_{reach}$ for our example program. We first obtain the post-condition after applying the transition relation of the program once.

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

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

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

We repeat the application step once again.

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

So far, by iteratively applying the post-condition function to $\varphi_{init}$ 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, $\varphi_{reach}$ is the above disjunction. □

Let $\rho_1$ and $\rho_2$ be formulas over $V$ and $V'$. We define a *relational composition* function $\circ$ as follows.

$$\rho_1 \circ \rho_2 \ = \ \exists V'' : \rho_1[V''/V'] \wedge \rho_2[V''/V] \tag{0.8}$$

Given $\rho_{\mathcal{R}}$ and a natural number $n$, we define an *n-time transition composition* $comp^n(\rho_{\mathcal{R}})$ as follows.

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

*Example 8.* For example, given the transition relations $\rho_1$, $\rho_2$, and the program variables $V = (pc, x, y, z)$ from our example program we obtain the following relational composition.

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

□

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

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

### *0.2.4 Inductive Safety and Termination Arguments*

An *inductive invariant* $\varphi$ contains the intial states and is closed under successors. 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 $\rho_{\mathcal{R}}$, i.e.,

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

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

*Example 9.* 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 7 and is shown below.

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

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.

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

□

An *inductive transition invariant* $\psi$ contains the restriction of the program transition relation to reachable states and is closed under relational composition with the program transition relation. Formally, given an inductive invariant $\varphi$ we require that an inductive transition invariant $\psi$ satisfies the following conditions:

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

A program terminates if there exist a finite number of well-founded relations over program states $WF_1, \ldots, WF_n$ whose union contains an inductive transition invariant, i.e., $\psi \models WF_1 \vee \ldots \vee WF_n$.

## 0.3 Abstract reachability computation

Computation of reachable program states requires iterative application of the post-condition function on the initial program states, see Equation (0.7). The iteration finishes when no new program states are discovered. Unfortunately, such an iteration process does not terminate in finite time.

*Example 10.* For example, 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 $\rho_2$ 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.,

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

□

Instead of computing $\varphi_{reach}$ we compute an over-approximation of $\varphi_{reach}$ 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 $\varphi_{reach}$, 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

$$\forall \varphi \ \forall \rho : post(\varphi, \rho) \models post^{\#}(\varphi, \rho) \ . \tag{0.9}$$

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

$$\forall \varphi : \varphi \models \alpha(\varphi) \ . \tag{0.10}$$

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

$$post^{\#}(\varphi, \rho) \ = \ \alpha(post(\varphi, \rho)) \tag{0.11}$$

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

$$\begin{aligned} \varphi^{\#}_{reach} &= \alpha(\varphi_{init}) \vee \\ &\quad post^{\#}(\alpha(\varphi_{init}), \rho_{\mathcal{R}}) \vee \\ &\quad post^{\#}(post^{\#}(\alpha(\varphi_{init}), \rho_{\mathcal{R}}), \rho_{\mathcal{R}}) \vee \ldots \\ &= \textstyle\bigvee_{i \geq 0} (post^{\#})^i(\alpha(\varphi_{init}), \rho_{\mathcal{R}}) \end{aligned} \tag{0.12}$$

The following lemma formalizes our over-approximation based reachability computation.

**Lemma 1.** $\varphi_{reach} \models \varphi^{\#}_{reach}$

### *0.3.1 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 $\mathit{Preds} = \{p_1, \ldots, p_n\}$. Then, we define an over-approximation of $\varphi$ that is represented using *Preds* as follows.

$$\alpha(\varphi) = \textstyle\bigwedge \{p \in \mathit{Preds} \mid \varphi \models p\} \tag{0.13}$$

*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.

| | $at\_\ell_1$ | $at\_\ell_2$ | $at\_\ell_3$ | $at\_\ell_4$ | $at\_\ell_5$ | $y \geq z$ | $x \geq y$ |
|---|---|---|---|---|---|---|---|
| $at\_\ell_2 \wedge y \geq z \wedge x + 1 \leq y$ | $\not\models$ | $\models$ | $\not\models$ | $\not\models$ | $\not\models$ | $\models$ | $\not\models$ |

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

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

If the set of predicates is empty then the result of applying predicate abstraction is *true*. For example, for $\mathit{Preds} = \emptyset$ we obtain

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

If no predicates in *Preds* is entailed the resulting abstraction is *true* as well. For example, for $\mathit{Preds} = at\_1, \dots, at\_3$ we have

$$\alpha(at\_\ell_5) = \bigwedge \emptyset = \mathit{true} \ .$$

□

The predicate abstraction function in Equation (0.13) approximates $\varphi$ 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 0.12. Let $\mathit{Preds} = \{\mathit{false}, at\_\ell_1, \dots, at\_\ell_5, y \geq z, x \geq y\}$. First, let $\varphi_1$ be the over-approximation of the set of initial states $\varphi_{init}$:

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

We apply $\mathit{post}^{\#}$ on $\varphi_1$ wrt. each program transition and obtain

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

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

Now we apply program transitions on $\varphi_2$ using $\mathit{post}^{\#}$. The application of $\rho_1$, $\rho_4$, and $\rho_5$ on $\varphi_2$ results in *false* for the following reason. $\varphi_2$ requires $at\_\ell_2$, but the transition relations $\rho_1$, $\rho_4$, and $\rho_5$ are applicable if either $at\_\ell_1$ or $at\_\ell_3$ holds. For $\rho_2$ we obtain

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

The resulting set above is equal to $\varphi_2$ and, therefore, is discarded, since we are already exploring states reachable from $\varphi_2$. For $\rho_3$ we obtain

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

We compute an over-approximation of the set of states that are reachable from $\varphi_3$ by applying $\mathit{post}^{\#}$. The transitions $\rho_1$, $\rho_2$, and $\rho_3$ results in *false* due to an inconsistency caused by the program counter valuations in $\varphi_3$ and the respective transition relations. For the transition $\rho_4$ we obtain

```
   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 $\rho \in \mathcal{R}$ 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. 0.2** Algorithm AbstReach for abstract reachability computation wrt. a given finite set of predicates.

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

For the transition $\rho_5$, which corresponds to the assertion violation, we obtain

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

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 $\varphi^{\#}_{reach} \wedge at\_\ell_5 \models false$ the program is safe. □

### *0.3.2 Algorithm* AbstReach

We combine the characterization of abstract reachability using Equation (0.12) with the predicat abstraction function given in Equation (0.13) and obtain an algorithm AbstReach for computing $ReachStates^{\#}$. The algorithm is shown in Figure 0.2.

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 0.2. First, we construct the abstraction function $\alpha$ according to Equation (0.13), and then use it to construct an over-approximation $post^{\#}$ of the post-condition function according to Equation (0.11). We initialize $ReachStates^{\#}$ with an over-approximation of the initial program states, which corresponds to the first disjunct in Equation (0.12). Since

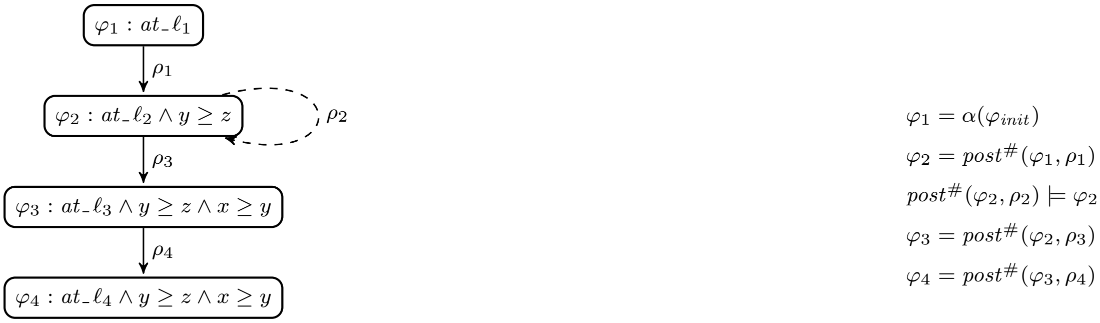

$\varphi_1 : at\_\ell_1$
$\rho_1$
$\varphi_2 : at\_\ell_2 \wedge y \geq z$
$\rho_2$
$\rho_3$
$\varphi_3 : at\_\ell_3 \wedge y \geq z \wedge x \geq y$
$\rho_4$
$\varphi_4 : at\_\ell_4 \wedge y \geq z \wedge x \geq y$
$\varphi_1 = \alpha(\varphi_{init})$
$\varphi_2 = post^{\#}(\varphi_1, \rho_1)$
$post^{\#}(\varphi_2, \rho_2) \models \varphi_2$
$\varphi_3 = post^{\#}(\varphi_2, \rho_3)$
$\varphi_4 = post^{\#}(\varphi_3, \rho_4)$

**Fig. 0.3** Applying AbstReach on the program in Figure 0.1 and the set of predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq y\}$. The nodes $\varphi_1$, …, $\varphi_4$ 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 $\varphi_2$.

the initial states do not have any 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 (0.12) 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 $\varphi$, 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 $post^{\#}$ wrt. each program transition, say $\rho$, on $\varphi$. Let $\varphi'$ be the result of such an application. We add $\varphi'$ to $ReachStates^{\#}$ if $\varphi'$ 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 $\varphi'$ and the disjunction of all formulas in $ReachStates^{\#}$. Often, there is a formula $\psi$ in $ReachStates^{\#}$ such that $\varphi' \models \psi$. In case that $\varphi$ is added to $ReachStates^{\#}$, we record that $\varphi$ was computed by applying $\rho$ on $\varphi$ by adding a tuple $(\varphi, \rho, \varphi')$ to *Parent*. Finally, $\varphi'$ is put on the worklist.

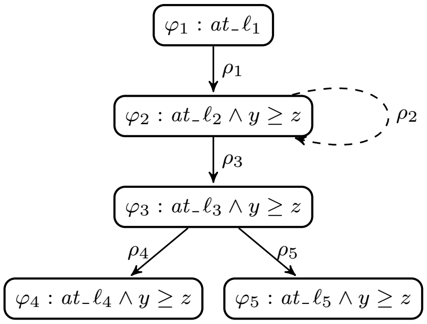
The loop execution terminates after a finite number of steps, since the range of $post^{\#}$ is finite (and is of size $2^n$ where $n$ is the size of *Preds*). The disjunction of formulas in $ReachStates^{\#}$ is logically equivalent to $\varphi^{\#}_{reach}$.

*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 0.3 provides a pictorial illustration. Example 12 provides details on computed over-approximations of post-conditions.

After constructing $\alpha$ and $post^{\#}$ for the given predicates, we compute $\varphi_1 = (at\_\ell_1)$ and put it into $ReachStates^{\#}$ and into *Worklist*. See the node $\varphi_1$ in Figure 0.3.

During the first loop iteration, we choose $\varphi_1$ to be the element taken from the worklist. Now we compute $post^{\#}$ wrt. each program transition. For $\rho_1$ 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 $\varphi_1$ and $\varphi_2 \not\models \varphi_1$. Hence, $\varphi_2$ is added to $ReachStates^{\#}$. As a result, the tuple $(\varphi_1, \rho_1, \varphi_2)$ is added to *Parent* and $\varphi_2$ becomes a worklist item. See the node $\varphi_2$ as well as the edge between $\varphi_1$ and $\varphi_2$ in Figure 0.3. We continue with applying program transitions on $\varphi_1$. For $\rho_2$ we obtain $post^{\#}(\varphi_1, \rho_2) = false$. Since $false \models \bigvee ReachStates^{\#}$ there is no addition to $ReachStates^{\#}$. Similarly, applying $\rho_3$, …, $\rho_5$ 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 $\varphi_2$ from the worklist. When applying $post^{\#}$ on $\varphi_2$ only $\rho_2$ and $\rho_3$ 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 $\varphi_2$ and hence $\bigvee ReachStates^{\#}$, nothing is added to $ReachStates^{\#}$ and we proceed directly with $\rho_3$. 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 $\varphi_3$


$\varphi_1 : at\_\ell_1$
$\rho_1$
$\varphi_2 : at\_\ell_2 \wedge y \geq z$
$\rho_2$
$\rho_3$
$\varphi_3 : at\_\ell_3 \wedge y \geq z$
$\rho_4$
$\rho_5$
$\varphi_4 : at\_\ell_4 \wedge y \geq z$
$\varphi_5 : at\_\ell_5 \wedge y \geq z$

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

**Fig. 0.4** Abstract reachability computation with $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z\}$.

to $ReachStates^{\#}$ and *Worklist*, while $(\varphi_2, \rho_3, \varphi_3)$ is recorded in *Parent*. See the node $\varphi_3$ as well as the edge between $\varphi_2$ and $\varphi_3$ in Figure 0.3.

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 $\varphi_3$ from the worklist. After computing $\varphi_4$ by applying $\rho_4$ and discovering that $\varphi_4 \not\models \bigvee ReachStates^{\#}$, we add $\varphi_4$ following the algorithm. See the node $\varphi_4$ as well as the edge between $\varphi_3$ and $\varphi_4$ in Figure 0.3. Since all other program transition yield *false* we proceed with the next iteration.

The fourth loop iteration removes $\varphi_4$ 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 $Parent = \{(\varphi_1, \rho_1, \varphi_2), (\varphi_2, \rho_3, \varphi_3), (\varphi_3, \rho_4, \varphi_4)\}$. □

Monotonicity:

$$\forall \varphi_1\ \forall \varphi_2 : (\varphi_1 \models \varphi_2) \rightarrow (\alpha(\varphi_1) \models \alpha(\varphi_2))$$

## 0.4 Refining abstraction for reachability

The algorithm AbstReach requires a set of predicates in order to compute an over-approximation of the reachable program states. Finding the right set of predicates that yields a sufficiently precise over-approximation is a difficult task.

### *0.4.1 Analysis of counterexample paths*

*Example 14.* In Example 13, the provided set of predicates was adequate for proving program safety. Omitting just one predicate, e.g., provide the predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z\}$ 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 0.4, we have $\varphi_5 \wedge \varphi_{err} \not\models false$. That is, AbstReach fails to prove the property without the predicate $x \geq y$.

We analyse the reason for the excessive over-approximation. Figure 0.4 shows that the *Parent* relation records a sequence of three steps leading to the computation of $\varphi_5$. First, we apply $\rho_1$ to $\varphi_1$ and compute $\varphi_2$. Then, $\varphi_3$ is obtained by applying $\rho_3$ to $\varphi_2$. Finally, $\rho_5$ is applied to $\varphi_3$ and results in $\varphi_5$. Thus, we note that the sequence of program transitions $\rho_1$, $\rho_3$, and $\rho_5$ determined $\varphi_5$. We refer to this sequence as a counterexample path. Using this path and the functions $\alpha$ 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, $\varphi_5$ 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

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

Hence, by executing the program transitions $\rho_1$, $\rho_3$, and $\rho_5$ 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 $\rho_1$, $\rho_3$, and $\rho_5$. Concretely, we need a refined abstraction function $\alpha$ 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:

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

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

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

The over-approximation given by $\psi_1, \dots, \psi_4$ 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.

| $\psi_1$ | $\psi_2$ | $\psi_3$ | $\psi_4$ |
|---|---|---|---|
| $at\_\ell_1$ | $at\_\ell_2 \wedge y \geq z$ | $at\_\ell_3 \wedge x \geq z$ | $false$ |

Due to a particularly useful property of predicate abstraction function and the post-condition function, which we discuss later in this section, we can use the obtained solution to refine $\alpha$ in the following way. By adding $\psi_1, \dots, \psi_4$ to the set of predicates *Preds* we guarantee that the resulting $\alpha$ and $post^{\#}$ are sufficiently precise to show that no error state is reachable along the path $\rho_1$, $\rho_3$, and $\rho_5$. Formaly, we obtain

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

**Fig. 0.5** Path computation.

```
function FeasiblePath
input
  $\rho_1 \dots \rho_n$ - path
begin
  $\varphi$ := $post(\varphi_{init}, \rho_1 \circ \dots \circ \rho_n)$
  if $\varphi \wedge \varphi_{err} \not\models false$ then
    return $true$
  else
    return $false$
end
```

**Fig. 0.6** Feasibility of a path.

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

□

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

The algorithms MakePath is shown in Figure 0.5. It takes as input a rechable abstract state $\psi$ together with a *Parent* relation. We view *Parent* as a tree where $\psi$ occurs as a node. MakePath outputs a sequence of program transitions that labels the tree edges connecting $\psi$ 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 15.* For our example tree in Figure 0.4 we construct the path by making a call MakePath($\varphi_5$, *Parent*). Then, *path* is extended with the transitions $\rho_5$, $\rho_3$, and $\rho_1$ 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 RefinePath
input
  $\rho_1 \ldots \rho_n$ - path
begin
  $\varphi_0, \ldots, \varphi_n$ := compute such that
      $(\varphi_{init} \models \varphi_0) \wedge$
      $(post(\varphi_0, \rho_1) \models \varphi_1) \wedge \ldots \wedge (post(\varphi_{n-1}, \rho_n) \models \varphi_n) \wedge$
      $(\varphi_n \wedge \varphi_{err} \models false)$
  return $\{\varphi_0, \ldots, \varphi_n\}$
end
```

**Fig. 0.7** Counterexample guided discovery of predicates.

The algorithms FeasiblePath is shown in Figure 0.6. 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 16.* 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

$$\rho_1 \circ \rho_3 \circ \rho_5 = \mathit{false}\ .$$

Hence, FeasiblePath sets $\varphi$ to *false* and then returns *false*.

The algorithms RefinePath is shown in Figure 0.7. 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 $\varphi_n \wedge \varphi_{err} \models \mathit{false}$. 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 $\alpha$ and $post^{\#}$ guarantee that

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

Here, we intensionally omit details of a particular algorithm for finding $\varphi_0, \ldots, \varphi_n$ that satisfy the above conditions. We discuss possible alternatives in Section 0.5.

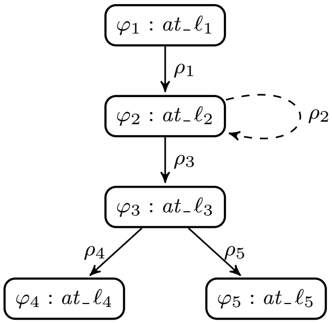
*Example 17.* As discussed in Example 14, the application of RefinePath on $\rho_1 \rho_3 \rho_5$ yields a sequence of sets of states that can refine the abstraction to become sufficiently precise at least for dealing with the considered path. □

### *0.4.2 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 0.8.

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

**Fig. 0.8** Predicate abstraction and refinement loop.



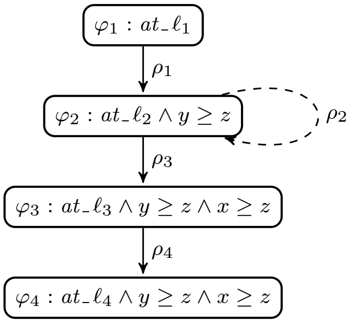
$$\varphi_1 = \alpha(\varphi_{init})$$
$$\varphi_2 = post^{\#}(\varphi_1, \rho_1)$$
$$post^{\#}(\varphi_2, \rho_2) \models \varphi_2$$
$$\varphi_3 = post^{\#}(\varphi_2, \rho_3)$$
$$\varphi_4 = post^{\#}(\varphi_3, \rho_4)$$
$$\varphi_5 = post^{\#}(\varphi_3, \rho_5)$$

**Fig. 0.9** Abstract reachability computation with $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5\}$.

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 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 $\psi$ in $ReachStates^{\#}$ that witnesses the intersection with the error states and use $\psi$ 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 $\psi$. 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*.

*Example 18.* We illustrate AbstRefineLoop using our example program from Figure 0.1. To make the illustration more vivid, we assume that $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5\}$ 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 0.9. In this figure, *Parent* is denoted by solid arrows that connect the formulas. We observe that $\varphi_5$ has a non-empty intersection with $\varphi_{err}$, hence we proceed by setting $\psi$ to $\varphi_5$. By applying MakePath we obtain $path = \rho_1\rho_3\rho_5$. At the next step, FeasiblePath reports


$\varphi_1 : at\_\ell_1$
$\rho_1$
$\varphi_2 : at\_\ell_2 \wedge y \geq z$
$\rho_2$
$\rho_3$
$\varphi_3 : at\_\ell_3 \wedge y \geq z \wedge x \geq z$
$\rho_4$
$\varphi_4 : at\_\ell_4 \wedge y \geq z \wedge x \geq z$

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

**Fig. 0.10** Applying AbstReach on the program in Figure 0.1 and the set of predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq z\}$.

that this path is not feasible, hence we proceed with the abstraction refinement. RefinePath discoveres that the predicates $y \geq z$ and $x \geq z$ are sufficient to refine the abstraction such that *path* is no longer leading 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 0.10 for the obtained set $ReachStates^{\#}$ and relation *Parent*. We observe that each formula in $ReachStates^{\#}$ has an empty intersection with $\varphi_{err}$, hence AbstRefineLoop reports that the program is correct. □

## 0.5 Solving refinement constraints

The algorithm RefinePath in Figure 0.7 takes as input an infeasible sequence of program transitions $\rho_1 \ldots \rho_n$ and computes sets of states $\varphi_0, \ldots, \varphi_n$ satisfying the following conditions.

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

Since $\rho_1 \ldots \rho_n$ 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.

### *0.5.1 Least solution*

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

$$\begin{aligned}\varphi_0 &= \varphi_{init} \\ \varphi_1 &= post(\varphi_0, \rho_1) \\ \dots \\ \varphi_n &= post(\varphi_{n-1}, \rho_n)\end{aligned} \tag{0.14}$$

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

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

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

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 19.* We illustrate how a least solution is computed using an example program shown in Figure 0.1.

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.

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

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. □

### 0.5.2 Greatest solution

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

$$wp(\varphi, \rho) \ = \ \forall V' : \rho \rightarrow \varphi[V'/V]. \tag{0.15}$$

For example, a transition $\rho_2$ from Figure 0.1 results in the following weakest precondition.

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

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

$$\begin{aligned}\varphi_n &= \neg\varphi_{err} \\ \varphi_{n-1} &= wp(\varphi_n, \rho_n) \\ &\dots \\ \varphi_0 &= wp(\varphi_1, \rho_1)\end{aligned} \tag{0.16}$$

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 20.* We illustrate how a greatest solution is computed using an example program shown in Figure 0.1.

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.

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

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 $\rho_1\rho_3\rho_5$. □

### 0.5.3 Intermediate solution using interpolation

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

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

$$\begin{aligned}&\varphi_0 = inter(\varphi_{init}, (\rho_1 \circ \ldots \circ \rho_n) \wedge \varphi_{err}[V'/V]) \\ &\varphi_1 = inter(post(\varphi_0, \rho_1), (\rho_2 \circ \ldots \circ \rho_n) \wedge \varphi_{err}[V'/V]) \\ &\dots \\ &\varphi_{n-1} = inter(post(\varphi_{n-2}, \rho_{n-1}), \rho_n \wedge \varphi_{err}[V'/V]) \\ &\varphi_n = inter(post(\varphi_{n-1}, \rho_n), \varphi_{err}[V'/V])\end{aligned} \tag{0.17}$$

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 21.* We illustrate how an intermediate solution is computed using an example program shown in Figure 0.1.

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.

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

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

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

□