Leaping Loops in the Presence of Abstraction

Thomas Ball[1], Orna Kupferman[2], and Mooly Sagiv[3]

[1] Microsoft Research, tball@microsoft.com
[2] Hebrew University, orna@cs.huji.ac.il
[3] Tel-Aviv University, msagiv@post.tau.ac.il

Abstract. Finite abstraction helps program analysis cope with the huge state space of programs. We wish to use abstraction in the process of error detection. Such a detection involves reachability analysis of the program. Reachability in an abstraction that under-approximates the program implies reachability in the concrete system. Under-approximation techniques, however, lose precision in the presence of loops, and cannot detect their termination. This causes reachability analysis that is done with respect to an abstraction to miss states of the program that are reachable via loops. Current solutions to this loop-termination challenge are based on fair termination and involve the use of well-founded sets and ranking functions.

In many cases, the concrete system has a huge, but still finite set of states. Our contribution is to show how, in such cases, it is possible to analyze termination of loops without refinement and without well-founded sets and ranking functions. Instead, our method is based on conditions on the structure of the graph that corresponds to the concrete system — conditions that can be checked with respect to the abstraction. We describe our method, demonstrate its usefulness and show how its application can be automated by means of a theorem prover.

1 Introduction

Finite abstraction (such as predicate or Boolean abstraction [7, 2]) helps program analysis cope with the huge state space of programs. Finite abstraction is helpful for proving properties of programs but less helpful for proving the presence of errors. The reason, as we demonstrate below, is that reachability analysis that is done with respect to an abstraction misses states of the program that are reachable via loops.

```
procedure simple (int n)
  int x:=0;
  while (x < n) do x:=x+1;
  while (x < 2n) do x:=x+2;
  while (x < 3n) do x:=x+3;
  assert false
```

Fig. 1. The procedure `simple`.

Consider the procedure `simple` appearing in Figure 1. The procedure is indeed simple and it increments the value of a variable x in a deterministic manner. It is not hard to see that the value of x eventually exceeds the value $3n$ and that the single execution of the procedure eventually reaches the failing assertion. Most counterexample-driven refinement methods, however, will generate a predicate for each loop iteration, quickly overwhelming the ability of their analysis engines to cope with the resulting state space explosion.

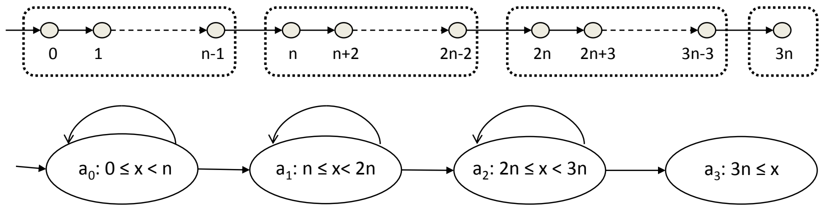

Fig. 2. The concrete state space of the procedure `simple` and its abstraction.

To see the problem in more detail, consider Figure 2, where we describe the state space of the procedure `simple`[4] and its abstraction according to the predicates $\{0 \leq x < n, n \leq x < 2n, 2n \leq x < 3n, 3n \leq x\}$. Since the abstraction over-approximates the transitions in the concrete system, and over-approximating transitions are not closed under transitivity, we cannot conclude, based on the abstraction, that a concrete state corresponding to a_3 is reachable from the a concrete state corresponding to a_0. Formally, the abstraction is a modal transition system (MTS) [11] in which all the transitions are *may* transitions. According to the three-valued semantics for modal transition systems [9], the property "exists a path in which $3n \leq x$" has truth value "unknown" and the abstraction should be refined. Since the three-valued abstraction gives a definite true value for reachability properties only if they hold along *must* transitions, the only refinement that would work bisimulates the concrete system. Augmenting MTSs with hyper-must transitions [12, 14] does not help in this setting either (and is orthogonal to the contribution we describe here).

Proving reachability along loops is a long-standing challenging problem in program abstraction. Recently, significant progress has been made by automatically proving termination [13, 5, 6, 4]. The main idea is to synthesize ranking functions proving well foundedness. However, these techniques require the generation of rank functions and/or are not suitable for proving that there exists a trace leading to a certain configuration in non-deterministic systems, which is a goal of our work.

In many cases (in particular, in all realistic implementations of software with variables over unbounded domains), the concrete system has a huge, but still finite set of states. Our contribution is to show how, in such cases, it is possible to analyze reachability in the concrete system without refinement of loops and without well-founded sets and ranking. Instead, our method is based on conditions on the structure of the graph

[4] The concrete values in Figure 2 correspond to the case $n = 0 \bmod 6$. Otherwise, the maximal value in a_1 may not be $2n - 2$, and similarly for a_2 and $3n - 3$.

that corresponds to the concrete system — conditions that can be checked automatically with respect to the abstraction.

Figure 3 illustrates the idea of our method, which is to replace the may transitions to and from an abstract state a by *must* transitions to an *entry port* for a and from an *exit port* for a, and to replace the intermediate may transition by a sequence of must transitions from the entry port to the exit port. Essentially, this is done by checking conditions that guarantee that the transitions of the concrete system embody a connected acyclic graph that has the entry port as its source and has the exit port as its sink. Finiteness of the set of concrete states associated with the abstract state then guarantees the finiteness of this graph. The checks we do, as well as the declaration of the entry and the exit ports, are automatic, refer to the abstract system, and are independent of the size of the concrete system. While our conditions are sufficient but not necessary, they are expected to hold in many cases.

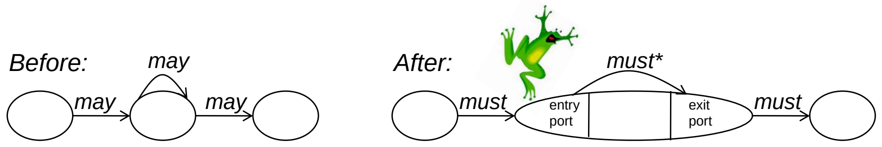

Fig. 3. Applying our method.

An approach similar to ours is taken in [10], where loop leaping is also performed without well-founded sets. Like our approach, the algorithm in [10] is based on symbolic reasoning about the concrete states associated with the loop. The conditions that the algorithm in [10] imposes, however, are different, and the algorithm is much more complicated. Essentially, loop detection along an abstract path $a_1, \ldots, a_n$ is reduced in [10] to the satisfiability of a propositional formula that specifies the existence of locations a_i and a_j along the path such that a_i is reachable from a_j and a_j is reachable from a_i. The size of the formula is quadratic in size of the concrete state space. Our conditions, on the other hand, are independent of the size of the concrete state space, and are much simpler. As we argue in the paper, the conditions we give are likely to be satisfied in many common settings.

2 Preliminaries

Programs and Concrete Transition Systems Consider a program P. Let X be the set of variables appearing in the program and variables that encode the program counter (pc), and let D be the domain of all variables (for technical simplicity, we assume that all variables are over the same domain). We model P by a concrete transition system in which each state is labeled by a valuation in $D^{|X|}$.

A *concrete transition system* (CTS) is a tuple $C = \langle S_C, I_C, \longrightarrow_C \rangle$, where S_C is a (possibly infinite) set of states, $I_C \subseteq S_C$ is a set of initial states, $\longrightarrow_C \subseteq S_C \times S_C$ is a total transition relation. Given a concrete state $c \in S_C$, let $s(c)$ denote the successor states of c; that is, $s(c) = \{c' \in S_C \mid c \longrightarrow_C c'\}$, and let $p(c)$ denote the predecessor states of c; that is, $p(c) = \{c' \in S_C \mid c' \longrightarrow_C c\}$. Let $c {\longrightarrow_C}^* c'$ denote that state c' is reachable from state c via a path of transitions.

A CTS is *deterministic* if every state has a single successor. A CTS is *reverse-deterministic* if every state has a single predecessor. Nondeterminism in concrete systems is induced by internal or external nondeterminism, as well as resource allocation and built-in abstractions in the programs they model.

Predicate Abstraction Let $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ be a set of predicates (formulas of first-order logic) over the program variables X. Given a program state c and formula ϕ, let $c \models \phi$ denote that formula ϕ is true in state c (c is a model of ϕ). For a set $a \subseteq \Phi$ and an assignment $c \in D^{|X|}$, we say that c *satisfies* a iff $c \models \bigwedge_{\phi_i \in a} \phi_i$.

In predicate abstraction, we merge a set of concrete states into a single abstract state, which is defined by means of a subset of the predicates. Thus, an abstract state is given by a set of predicates $a \subseteq \Phi$.[5] We sometimes represent a by a formula, namely the conjunction of predicates in a. For example, if $a = \{(x \geq y), (0 \leq x < n)\}$ then we also represent a by the formula $(x \geq y) \wedge (0 \leq x < n)$. We define the set of concrete states corresponding to a, denoted $\gamma(a)$, as all the states c that satisfy a; that is, $\gamma(a) = \{c \mid c \models a\}$.

May and Must Transitions Given a concrete transition system and its (predicate) abstraction via a set of predicates Φ, its *modal transition system* (MTS) contains three kinds of abstract transitions between abstract states a and a' ($a, a' \subseteq \Phi$, and we assume that Φ is clear from the context):

- $may(a, a')$ if there is $c \in \gamma(a)$ and a $c' \in \gamma(a')$, such that $c \longrightarrow_C c'$.
- $must^+(a, a')$ only if for every $c \in \gamma(a)$, there is $c' \in \gamma(a')$ such that $c \longrightarrow_C c'$.
- $must^-(a, a')$ only if for every $c' \in \gamma(a')$, there is $c \in \gamma(a)$ such that $c \longrightarrow_C c'$.

Must transitions are closed under transitivity, and can therefore be used to prove reachability in the concrete system. Formally, if there is a sequence of $must^+$-transitions from a to a' (denoted by ${must^+}^*(a, a')$) then for all $c \in \gamma(a)$, there is $c' \in \gamma(a')$ such that $c {\longrightarrow_C}^* c'$. Dually, if there is a sequence of $must^-$-transitions from a to a' (denoted by ${must^-}^*(a, a')$) then for all $c' \in \gamma(a')$, there is $c \in \gamma(a)$ such that $c {\longrightarrow_C}^* c'$. On the other hand, may transitions are not transitive. Indeed, it may be the case that $may(a, a'), may(a', a'')$, and still for all $c \in a$ and $c'' \in a''$, we have $c {\not\longrightarrow_C}^* c''$.

Let us go back to the procedure `simple` and its abstraction in Figure 2. Since every concrete state in a_3 has a predecessor in a_2, we have that $must^-(a_2, a_3)$. On the other hand, all the other transitions in the abstraction are may transitions. As such, we cannot use the abstraction in order to conclude that the failing statement is reachable from the initial state. We want to detect such reachability, and we want to do it without well-founded orders and without refining the abstraction further!

[5] In the full generality of predicate abstraction, an abstract state is represented by a set of sets of predicates (that is a, disjunction of conjunction of predicates). All our results hold for the more general setting.

Weakest Preconditions and Strongest Postconditions In many applications of predicate abstraction, Φ includes a predicate for the program counter. Accordingly, each abstract state is associated with a location of the program, and thus it is also associated with a statement. For a statement s and a predicate e over X, the *weakest precondition* $\mathrm{WP}(s, e)$ and the *strongest postcondition* $\mathrm{SP}(s, e)$ are defined as follows [8]:

- The execution of s from every state that satisfies $\mathrm{WP}(s, e)$ results in a state that satisfies e, and $\mathrm{WP}(s, e)$ is the weakest predicate for which the above holds.
- The execution of s from a state that satisfies e results in a state that satisfies $\mathrm{SP}(s, e)$, and $\mathrm{SP}(s, e)$ is the strongest predicate for which the above holds.

For example, in the procedure `simple`, we have $\mathrm{WP}(x := x + 2, n \leq x < 2n) = n \leq x + 2 < 2n$, $\mathrm{SP}(x := x + 2, n \leq x < 2n) = n + 2 \leq x < 2n + 2$.

Must transitions can be computed automatically using weakest preconditions and strongest postconditions. Indeed, statement s induces the transition $must^+(a, a')$ iff $a \Rightarrow \mathrm{WP}(s, a')$, and induces the transition $must^-(a, a')$ iff $a' \Rightarrow \mathrm{SP}(s, a)$.

We sometimes use also the Pre predicate. For a statement s and a predicate e over X, the execution of s from a state that satisfies $\mathrm{Pre}(s, e)$ may result in a state that satisfies e. Formally, $\mathrm{Pre}(s, e) = \neg\mathrm{WP}(s, \neg e)$.

3 Leaping Loops

Unfortunately, an abstraction of loops usually results in may transitions. As discussed above, may transitions are not closed under transitivity, thus abstraction methods cannot cope with reachability of programs with loops. In this section we describe our method for coping with loops.

An *entry port* of an abstract state a is a predicate e_a such that $\gamma(e_a) \subseteq \gamma(a)$ and for all $c_e \in \gamma(e_a)$, either c_e is initial or $p(c_e) \setminus \gamma(a) \neq \emptyset$. That is, every concrete state c_e represented by entry port e_a is inside a and either c_e is initial or some predecessor of c_e lies outside a.

Dually, an *exit port* of an abstract state a is a predicate x_a such that $\gamma(x_a) \subseteq \gamma(a)$ and for all $c_x \in \gamma(x_a)$, we have that $s(c_x) \setminus \gamma(a) \neq \emptyset$. That is, every concrete state c_x represented by exit port x_a is in a and some successor of c_x lies outside a.

In Section 4.1, we describe how entry and exit ports can be calculated automatically be means of weakest preconditions and strongest postconditions. We now use entry and exit ports in order to reason about loops.

Theorem 1. *Consider an abstract state a. Let e_a and x_a be entry and exit ports of a such that all the following conditions hold:*

1. *$\gamma(a)$ is finite;*
2. *for all $c \in \gamma(a \wedge \neg x_a)$, we have that $|\ s(c) \cap \gamma(a)\ | \leq 1$. That is, every concrete state in $\gamma(a \wedge \neg x_a)$ has at most one successor in $\gamma(a)$.*
3. *$must^-(a \wedge \neg x_a, a \wedge \neg e_a)$. That is, every concrete state in $\gamma(a \wedge \neg e_a)$ has a predecessor in $\gamma(a \wedge \neg x_a)$.*

Then, $must^{-*}(e_a, x_a)$. That is, for all $c' \in \gamma(x_a)$, there is $c \in \gamma(e_a)$ such that $c \longrightarrow_C{}^* c'$.

Note that Conditions 1-3 imply that e_a cannot be empty (unless x_a is empty, in which case the theorem holds trivially).

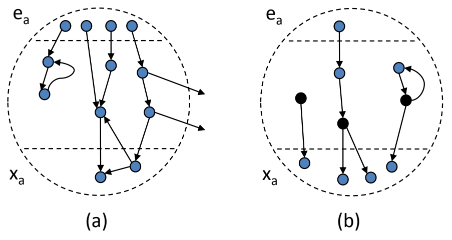

Fig. 4. Inside an abstract state.

The proof of Theorem 1 is based on constructing a DAG in which all states are reachable from the source. The finiteness of $\gamma(a)$ then implies that source vertices of the DAG are contained in $\gamma(e_a)$. Note that the DAG induces a well-founded order on the states of $\gamma(a)$. The well-founded order, however, is hidden in the proof and the user does not have to provide it.

Before we prove the theorem, let us give some intuition and an example to its application. Figure 4(a) illustrates the intuition underlying Theorem 1. The large dashed circle represents the abstract state a with entry port e_a and exit port x_a. The grey nodes represent concrete states that are consistent with the theorem. Every grey node that is not in the exit port has at most one successor in a (but may have arbitrarily many successors outside a). Every grey node in $\gamma(a \wedge \neg e_a)$ has a predecessor in $a \wedge \neg x_a$ (and may have more than one predecessor). Note that the conditions permit cycles in the concrete state space, as shown on the left of the figure.

The black nodes in Figure 4(b) illustrate configurations in the concrete state space that are not permitted by the theorem. We see that the conditions of the theorem rule out unreachable cycles, as well as non-determinism inside a. Finally, it is not permitted to have a state in $\gamma(a \wedge \neg e_a)$ that does not have predecessor in $\gamma(a \wedge \neg x_a)$.

Example 1. Consider the procedure `simple` from Figure 1 and its abstraction in Figure 2. The application of our method on the abstraction is described in Figure 5. The abstract state $a_0 : 0 \leq x < n$ has entry port $x = 0$ and exit port $x = n - 1$. The conditions of Theorem 1 hold for a_0 with these ports: first, as n is finite, so is $\gamma(a_0)$. Second, since the procedure is deterministic, each concrete state has a single successor. Finally, each concrete state except for $x = 0$ has a predecessor in a_0. We can therefore conclude that $must^{-*}(x = 0, x = n - 1)$. In a similar way, the conditions of the theorem hold for a_1 with entry port $x = n$ and exit port $2n - 2 \leq x < 2n$, and for a_2 with entry port $2n \leq x \leq 2n + 1$ and exit port $3n - 3 \leq x < 3n$. From this, we can conclude that $must^{-*}(x = n, 2n-2 \leq x < 2n)$ and $must^{-*}(2n \leq x \leq 2n+1, 3n-3 \leq x < 3n)$. Since, in addition, $must^{-}(x = n - 1, x = n)$, $must^{-}(2n - 2 \leq x < 2n, 2n \leq x \leq 2n+1)$, and $must^{-}(3n-3 \leq x < 3n, 3n \leq x)$, we can conclude, from the transitivity of $must^{-}$, that $must^{-*}(x = 0, 3n \leq x)$.

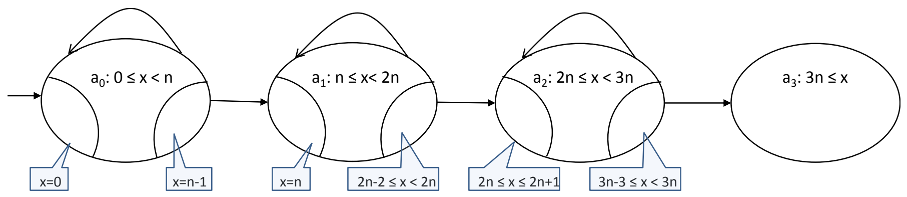

Fig. 5. Entry and exit ports in the abstraction of the procedure simple.

Proof: Assume that $\gamma(x_a)$ is not empty; otherwise, $must^{-*}(e_a, x_a)$ holds trivially, and we are done. For a directed acyclic graph (DAG) $G = \langle V, E \rangle$, let $source(G) = \{c \in V \mid \forall c' \in V, \neg E(c', c)\}$ be the set of vertices in V that do not have predecessors in G. We construct a sequence of DAGs $G_0, G_1, \ldots$, such that for all $k \geq 0$, the DAG $G_k = \langle V_k, E_k \rangle$ is such that $\gamma(x_a) \subseteq V_k \subseteq \gamma(a)$ ("the exit property") and every state in V_k is reachable from some state in $source(G_k)$ ("the reachability property"). In addition, for every $k \geq 0$, either $source(G_k) \subseteq \gamma(e_a)$ ("the entrance property"), or we can construct a DAG G_{k+1} that satisfies the exit and reachability properties and strictly contains G_k. Since $\gamma(a)$ is finite [Condition 1 of the theorem], the above implies we must eventually reach $k \leq |\gamma(a)|$ for which $source(G_k) \subseteq \gamma(e_a)$.

Note that since G_k is a subgraph of the concrete system, the reachability property implies that $must^{-*}(source(G_k), V_k)$. Thus, the existence of a DAG G_k that satisfies all three properties, implies that $must^{-*}(e_a, x_a)$, and we are done.

We define G_k by induction on k, which would be the height of the DAG. For the induction base, let $G_0 = \langle V_0, E_0 \rangle$ with $V_0 = \gamma(x_a)$ and $E_0 = \emptyset$. It is easy to see that G_0 satisfies both the exit and reachability properties. In particular, since $E_0 = \emptyset$, we have that $source(G_0) = V_0$, thus the reachability property is satisfied with empty paths. Note that since $\gamma(x_a)$ is not empty, so are V_0 and $source(G_0)$.

For the induction step, let $k \geq 0$ be such that G_k satisfies both the exit and reachability properties and does not satisfy the entrance property. We prove we can construct a DAG G_{k+1} that satisfies the exit and reachability properties and strictly contains G_k.

Let $S_k = source(G_k) \setminus \gamma(e_a)$. Note that since G_k does not satisfy the entrance property and $source(G_k)$ is not empty, the set S_k is not empty either. Since $must^-(a \wedge \neg x_a, a \wedge \neg e_a)$ [Condition 3 of the theorem] and $S_k \cap \gamma(e_a) = \emptyset$ [by the definition of S_k], then every state in S_k has a predecessor in $\gamma(a \wedge \neg x_a)$. Let $V'_k = \{p(c) \cap \gamma(a \wedge \neg x_a) \mid c \in S_k\}$. Note that $V'_k \neq \emptyset$. Let $V_{k+1} = V_k \cup V'_k$, and let $E_{k+1} = E_k \cup \{\langle c', c \rangle \mid c \in S_k$ and $c' \in p(c) \cap \gamma(a \wedge \neg x_a)\}$. Thus, G_{k+1} adds to G_k states in $\gamma(a \wedge \neg x_a)$ that have a transition to states in $source(G_k) \setminus \gamma(e_a)$, and it also adds the transitions from these states to the states in $source(G_k) \setminus \gamma(e_a)$.

Since $V_k \subseteq V_{k+1}$, and, by the induction hypothesis, $\gamma(x_a) \subseteq V_k$, then $\gamma(x_a) \subseteq V_{k+1}$, thus G_{k+1} satisfies the exit property.

We prove that V_k and V'_k are disjoint. Since, by the induction hypothesis, G_k is a DAG, this implies that G_{k+1} is a DAG too. Indeed, the edges we have added to G_k are edges from V'_k to V_k. Also, since $V'_k \neq \emptyset$, the fact that V_k and V'_k are disjoint implies that V_{k+1} strictly contains V_k.

For the case $k = 0$, we have $V_0 = \gamma(x_a)$. Since V'_k contains only states in $\gamma(a \wedge \neg x_a)$, then clearly $V_0 \cap V'_0 = \emptyset$. For the case $k > 0$, assume by way of contradiction that there is $c \in V'_k \cap V_k$. Since V'_k and $\gamma(x_a)$ are disjoint, the fact that $c \in V_k$ implies that there is $1 \leq j < k$ such that $c \in V'_j$. That is, j is the iteration in which c has joined G_k. Hence, there is $c' \in s(c) \cap \gamma(a)$. That is, $c' \in \gamma(a)$ is the successor of c that has caused its membership in V'_j. Since $c \in \gamma(a \wedge \neg x_a)$, then, by Condition 2 of the theorem, $|s(c) \cap \gamma(a)| \leq 1$. Therefore, it must be that $c' \in S_k$; indeed, c' is the only successor of c in $\gamma(a)$, and in order for c to be in V'_k, its single successor in $\gamma(a)$ has to be in S_k. Since, however, c' has a predecessor (that is c) in G_k, it is not in $source(G_k)$. Therefore, $c' \notin S_k$, and we have reached a contradiction.

It is left to prove that G_{k+1} satisfies the reachability property. By definition, $source(G_{k+1}) = V'_k$. Also, by the induction hypothesis, all states in V_k are reachable from some state in $source(G_k)$. Since each state in $source(G_k)$ is the successor of some state in V'_k, it follows that every state in V_{k+1} is reachable from some state in $source(G_{k+1})$. $\square$

We now state a similar theorem for a forward traversal.

Theorem 2. *Consider an abstract state a. Let e_a and x_a be entry and exit ports of a such that all the following conditions hold:*

1. *$\gamma(a)$ is finite.*
2. *for all $c \in \gamma(a \wedge \neg e_a)$, we have that $\mid p(c) \cap \gamma(a) \mid \leq 1$. That is, every concrete state in $\gamma(a \wedge \neg e_a)$ has at most one predecessor in $\gamma(a)$.*
3. *$must^+(a \wedge \neg x_a, a \wedge \neg e_a)$. That is, every concrete state in $\gamma(a \wedge \neg x_a)$ has a successor in $\gamma(a \wedge \neg e_a)$.*

Then, $must^{+*}(e_a, x_a)$. That is, for all $c \in \gamma(e_a)$, there is $c' \in \gamma(x_a)$ such that $c \longrightarrow_C{}^* c'$.

Proof: Assume that $\gamma(e_a)$ is not empty; otherwise, $must^{+*}(e_a, x_a)$ holds trivially, and we are done. For a DAG $G = \langle V, E\rangle$, let $sink(G) = \{c \in V \mid \forall c' \in V, \neg E(c, c')\}$ be the set of vertices in V that do not have successors in G. We construct a sequence of DAGs $G_0, G_1, \ldots$, such that for all $k \geq 0$, the DAG $G_k = \langle V_k, E_k\rangle$ is such that $\gamma(e_a) \subseteq V_k \subseteq \gamma(a)$ ("the entrance property") and every state in V_k can reach some state in $sink(G_k)$ ("the reachability property"). In addition, for every $k \geq 0$, either $sink(G_k) \subseteq \gamma(x_a)$ ("the exit property"), or we can construct a DAG G_{k+1} that satisfies the entrance and reachability properties and strictly contains G_k. Since $\gamma(a)$ is finite [Condition 1 of the theorem], the above implies we must eventually reach $k \leq |\gamma(a)|$ for which $sink(G_k) \subseteq \gamma(e_a)$.

Note that since G_k is a subgraph of the concrete system, the reachability property implies that $must^{+*}(V_k, sink(G_k))$. Thus, the existance of a DAG G_k that satisfies all three properties, implies that $must^{+*}(e_a, x_a)$, and we are done.

For the induction base, let $G_0 = \langle V_0, E_0\rangle$ with $V_0 = \gamma(e_a)$ and $E_0 = \emptyset$. It is easy to see that G_0 is a DAG and that it satisfies the entrance and reachability properties. In particular, since $E_0 = \emptyset$, we have that $sink(G_0) = V_0$, thus the reachability property is satisfied with empty paths. Note that since $\gamma(e_a)$ is not empty, so are V_0 and $sink(G_0)$.

For the induction step, let $k \geq 0$ be such that G_k satisfies entrance and reachability proeprties and does not satisfy the exit property. We prove we can construct a DAG G_{k+1} that satisfies the entrance and reachability properties and strictly contains G_k.

Let $S_k = sink(G_k) \backslash \gamma(x_a)$. Note that since G_k does not satisfy the exit property and $sink(G_k)$ is not empty, the set S_k is not empty either. Since $must^{+}(a \wedge \neg x_a, a \wedge \neg e_a)$ [Condition 3 of the theorem] and $S_k \cap \gamma(x_a) = \emptyset$ [by the definition of S_k], then every state in S_k has a successor in $\gamma(a \wedge \neg e_a)$. Let $V'_k = \{s(c) \cap \gamma(a \wedge \neg e_a) \mid c \in S_k\}$. Note that $V'_k \neq \emptyset$. Let $V_{k+1} = V_k \cup V'_k$, and let $E_{k+1} = E_k \cup \{\langle c, c'\rangle \mid c \in S_k$ and $c' \in s(c) \cap \gamma(a \wedge \neg e_a)\}$. Thus, G_{k+1} adds to G_k states in $\gamma(a \wedge \neg e_a)$ that are successors of states in $sink(G_k) \setminus \gamma(x_a)$, and it also adds the transitions from these states to the states in $sink(G_k) \setminus \gamma(x_a)$.

Since $V_k \subseteq V_{k+1}$, and, by the induction hypothesis, $\gamma(e_a) \subseteq V_k$, then $\gamma(e_a) \subseteq V_{k+1}$, thus G_{k+1} satisfies the entrance property.

We prove that V_k and V'_k are disjoint. Since, by the induction hypothesis, G_k is a DAG, this implies that G_{k+1} is a DAG too. Also, since $V'_k \neq \emptyset$, this implies that V_{k+1} strictly contains V_k.

For the case $k = 0$, we have $V_0 = \gamma(e_a)$. Since V'_k contains only states in $\gamma(a \wedge \neg e_a)$, then clearly $V_0 \cap V'_0 = \emptyset$. For the case $k > 0$, assume by way of contradiction that there is $c \in V'_k \cap V_k$. Since V'_k and $\gamma(e_a)$ are disjoint, the fact that $c \in V_k$ implies that there is $1 \leq j < k$ such that $c \in V'_j$. That is, j is the iteration in which c has joined G_k. Hence, there is $c' \in p(c) \cap \gamma(a)$. That is, $c' \in \gamma(a)$ is the predecessor of c that has caused its membership in V'_j. Since $c \in \gamma(a \wedge \neg e_a)$, then, by Condition 2 of the theorem, $|p(c) \cap \gamma(a)| \leq 1$. Therefore, it must be that $c' \in S_k$; indeed, c' is the only predecessor of c in $\gamma(a)$, and in order for c to be in V'_k, its single predecessor in $\gamma(a)$ has to be in S_k. Since, however, c' has a successor (that is c) in G_k, it is not in $sink(G_k)$. Therefore, $c' \notin S_k$, and we have reached a contradiction.

It is left to prove that G_{k+1} satisfies the reachability property. By definition, $sink(G_{k+1}) = V'_k$. Also, by the induction hypothesis, every state in V_k can reach some state in $sink(G_k)$. Since every state in $sink(G_k)$ has a successor in in V'_k, it follows that every state in V_{k+1} can reach a state in $sink(G_{k+1})$. □

As demonstrated in Example 1, the application of our method in a program with multiple loops requires the "gluing" of exit and entry ports. We refer to such ports as standard abstract states, thus gluing follows from the transitivity of $must^{-}$ and $must^{+}$ transitions.

Formally, we have the following.

Theorem 3. *Let a_1 and a_2 be abstract states with entry ports e_{a_1} and e_{a_2}, and exit ports x_{a_1} and x_{a_2}, respectively. Assume that for all $i \in \{1, 2\}$, we have that $\gamma(a_i)$ is finite, for all $c \in \gamma(a_i \wedge \neg x_{a_i})$, we have that $\mid s(c) \cap \gamma(a_i) \mid \leq 1$, and $must^{-}(a_i \wedge \neg x_{a_i}, a_i \wedge \neg e_{a_i})$, and in addition $must^{-*}(x_{a_1}, e_{a_2})$. Then, $must^{-*}(e_{a_1}, x_{a_2})$.*

Proof: By the conditions of the theorem, the conditions of Theorem 1 hold with respect to a_1 and a_2 and their entry and exit ports. Thus, $must^{-*}(e_{a_1}, x_{a_1})$ and $must^{-*}(e_{a_2}, x_{a_2})$. Since $must^{-*}(x_{a_1}, e_{a_2})$, the result follows from the transitivity of $must^{-}$ transitions. □

Since $must^+$ transitions are also transitive, Theorem 2 implies a "forward gluing" theorem dual to Theorem 3.

Below we discuss the conditions required for the application of Theorems 1 and 2 and describe more involved examples.

3.1 The $\gamma(a)$ finiteness assumption

Precondition (1) of Theorems 1 and 2 is that $\gamma(a)$ is finite. To see that the finiteness requirement is crucial, consider an abstract state over the whole numbers $a = (x \geq 0 \land y \geq 0)$, and assume that the statement executed in a is `while true do if y=0 then x:=x-1`. Let $e_a = (x \geq 0 \land y > 0)$ and $x_a = (x = y = 0)$. Note that e_a and x_a satisfy the conditions required from entry and exit ports: $\gamma(e_a) \subseteq \gamma(a)$, and $\gamma(e_a)$ may have predecessors not in $\gamma(a)$. Also, $\gamma(x_a) \subseteq \gamma(a)$, and the successor of the single concrete state in $\gamma(x_a)$ is not in $\gamma(a)$. Conditions (2) and (3) of Theorem 1 are satisfied: Each state in $\gamma(a)$ has a single successor, and all states in $\gamma(a \land \neg e_a) = \{\langle x, y\rangle : x \geq 0 \land y = 0\}$, have a predecessor in $\gamma(a \land \neg x_a)$. Still, we do not have $must^{-*}(e_a, x_a)$. Indeed, all states in $\gamma(e_a)$ satisfy $y \neq 0$ and therefore they have a self loop.

Note that while $\gamma(a)$ has to be finite, it is unbounded. Thus, for applications like detecting errors representing extreme out of bound resources, e.g., stack overflow, our method is applicable. Types like integers or reals have infinite domains. In practice, however, we run software on machines, where all types have finite representations. Thus, if for example, x is an integer and the abstract state $a : (x \geq 0)$ has an infinite $\gamma(a)$, we can view a as defined by the predicate $(0 \leq x \leq \max\_\mathrm{int})$, which is finite. Different machines have different policies for variables that go above their maximal or beyond their minimal values. It is possible to adjust the abstract system to account for these policies ("wrap around", error messages, etc.).

Another source of infiniteness are variables that the abstraction ignores. Consider for example a concrete state space over two integer variables, x and y. The abstract state $a : (1 \leq x \leq n)$ constrains x to have one of $|n|$ values but leaves y unconstrained, making $\gamma(a)$ infinite. Since, however, the behavior inside a is independent of y, its infiniteness is irrelevant to termination of a loop that traverses the values of x. This point, of coping with an abstraction that hides part of the variables is studied in [3]. Using *partitioned-must* transitions that are studied there, it is possible to apply Theorems 1 and 2 in settings in which there are finitely many equivalence classes in a partition of $\gamma(a)$ according to the value of x.

3.2 The determinization assumption

Consider the procedure `less_simple` described in Figure 6. A statement $s_1|s_2$ denotes a nondeterministic choice between statements s_1 and s_2. Thus, for example, in `x:=x+1|y:=x+1`, the procedure may either increment the value of x by 1 or assign $x+1$ to y. As in the procedure `simple`, the value of the variable x is incremented, but now the procedure may also assign values to the variable y, and the increments to x, as well as the failure assertion, depend on the relation between x and y.

```
procedure less_simple (int n)
  int x:=0; y:=0;
  x:=1; {y:=1|skip};
  while (x < n) do if x >= y+2 then y:=x else {x:=x+1|y:=x+1};
  if x >= y then assert false
```

Fig. 6. The procedure `less_simple`.

The behavior of the variables x and y is described in Figure 7. The figure also contains an abstraction of the procedure according to the predicates $\{(x = 0), (0 < x < n), (x \geq n), (y \leq x \leq y + 2), (x < y)\}$. We restrict the figure to states that are reachable along may transitions. Since the transition from a_1 to a_2 in the abstraction is a may transition, we cannot conclude that failure states are reachable from the initial state.

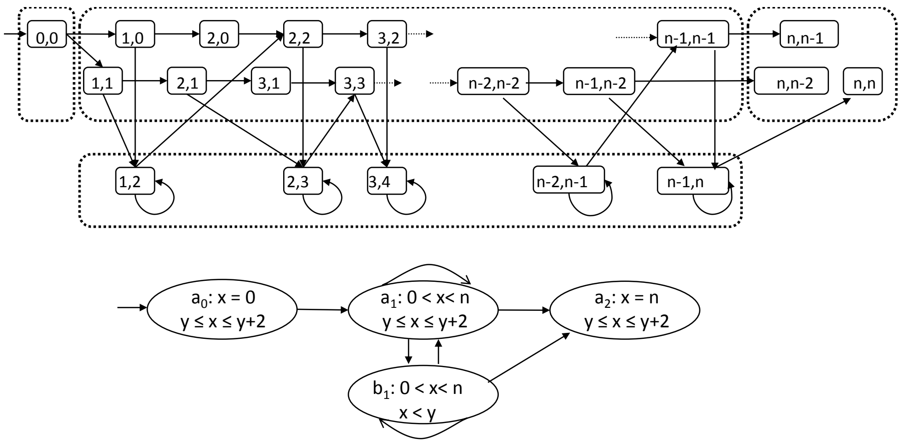

Fig. 7. An abstraction of the procedure `less_simple`.

Let us focus on the abstract state a_1, where $0 < x < n$ and $y \leq x \leq y + 2$. The predicate $(y \leq x = 1 \leq y + 2)$ is an entry port for a_1. Note that the state $x = y = 2$ is not in the entry port and still has a predecessor not in $\gamma(a_1)$, but an entry port need not be maximal. As an exit port, we take the predicate $(y \leq x = n - 1 \leq y + 2)$. Note that the transitions from some of the concrete states in a_1 (all these for which $y \leq x \leq y+1$) are nondeterministic. One of the nondeterministic choices, however, takes us out of a_1. Indeed, an attempt to use Theorem 1 without refining the predicate $0 < x < n$ to $y \leq x \leq y + 2$, $x < y$, and $x > y + 2$ fails. Note also that all the concrete states in $\gamma(a_1 \wedge \neg e_{a_1})$ have predecessors in $\gamma(a_1 \wedge \neg x_{a_1})$. Thus, $must^{-}(a_1 \wedge \neg x_{a_1}, a_1 \wedge \neg e_{a_1})$. The fact that some states (these in which $x = y$) have two predecessors, one of which is in b_1, does not violate the conditions of Theorem 1. By the theorem, all concrete states in the exit port are reachable from states in the entry port. Since, in addition, the error states $(x = n) \wedge (n - 2 \leq x \leq n - 1)$ are reachable from the exit port, and all states in the entry port are reachable from $x = y = 0$, we can conclude that some error states in `less_simple` are reachable from the initial state.

3.3 Nested loops

Proving termination is harder in the presence of nested loops. Our method, however, is applicable also to programs with nested loops. Consider the procedure `nested` in in Figure 8. Reasoning about the procedure `nested` with well-founded orders requires

```
procedure nested (int n)
  int y, x:=0;
  while x < n do
    x++; y:=0; while y < n do y++
  if y=n then assert false
```

Fig. 8. The procedure `nested`.

working with pairs in $\mathbb{N} \times \mathbb{N}$. Using our method, we can have a single abstract state $a : (0 \leq x, y \leq n)$, define the entry and exit ports to be $e_a = (x = y = 0)$ and $x_a = (x = y = n)$, respectively, and verify that the following conditions, of Theorem 2, hold: (1) $\gamma(a)$ is finite, (2) every concrete state in $\gamma(a \wedge \neg e_a)$ has at most one predecessor in $\gamma(a)$, and (3) every concrete state in $\gamma(a \wedge \neg x_a)$ has a successor in $\gamma(a \wedge \neg e_a)$.

Now, we can conclude that $must^{+*}(x = y = 0, x = y = n)$. Note that Theorems 1 and 2 can also be applied to more complicated variants of `nested` in which, for example, the increment to y depends on x. Complicated dependencies, however, may violate Condition (3) of the theorem, and the state a has to be refined in order for the condition to hold.

In general, our method is independent of the cause to the loop in the abstract state and can be applied to various cases like nested loops, recursive calls, and mutual recursive calls.

4 In Practice

In this section we discuss the implementation of our method and ways to use a theorem prover in order to automate it. We assume that the abstraction was obtained by predicate abstraction and that each abstract state is associated with a statement executed in all its corresponding concrete states.

We consider the following application: the user provides two abstract states a and a' and asks whether a' is *weakly reachable* from a'; that is, are there concrete states $c_0, c_1, \ldots, c_n$ such that $c_0 \in \gamma(a), c_n \in \gamma(a')$, and for all $0 \leq i < n$, we have

$c_i \longrightarrow_C c_{i+1}$. As discussed in Section 1, we have to check whether $must^{+*}(a, a')$ or $must^{-*}(a, a')$[6].

We start by considering a simpler mission, where the user also provides a path $a_1, a_2, \ldots, a_n$ in the abstract system such that $a = a_1$ and $a' = a_n$. Our method enters the picture in cases there is $1 < i < n$ such that a_i is associated with a loop, $may(a_{i-1}, a_i)$ or $may(a_i, a_{i+1})$. Then, as illustrated in Figure 3, we find entry and exit ports for a_i and check whether the conditions in Theorem 1 (or 2) are satisfied.

Below we describe how to automate both parts. We start with the detection of entry and exit ports.

4.1 Automatic calculation of ports along a path

For two abstract states a and a', and a statement s executed in a, we say that $e_{a'}$ is an *entry port for a' from a* if $\gamma(e_{a'}) \subseteq \gamma(a')$ and for all $c \in e_{a'}$, we have $p(c) \cap \gamma(a) \neq \emptyset$. Thus, $e_{a'}$ is an entry port and all its states have predecessors in a. Likewise, we say that x_a is an *exit port for a to a'* if $\gamma(x_a) \subseteq \gamma(a)$ and for all $c \in x_a$, we have $s(c) \cap \gamma(a') \neq \emptyset$. Thus, x_a is an exit port and all its states have successors in a'.

Lemma 1. *Consider two abstract states a and a'. Let s be the statement executed in a.*

- *$e_{a'}$ is an entry port for a' from a iff $e_{a'} \Rightarrow a' \wedge \mathrm{SP}(s, a)$.*
- *x_a is an exit port for a to a' iff $x_a \Rightarrow a \wedge \mathrm{Pre}(s, a')$.*

Proof: We start with entry ports. We start with the if direction. Let e'_a be such that $e_{a'} \Rightarrow a' \wedge \mathrm{SP}(s, a)$. We show that the two conditions for e'_a being an entry port are satisfied. First, since $e'_a \Rightarrow a'$, then clearly $\gamma(e'_a) \subseteq \gamma(a')$. For the second condition, consider a concrete state $c' \in \gamma(e'_a)$. Since c' satisfies $\mathrm{SP}(s, a)$, there is a state $c \in \gamma(a)$ such that c' is obtained from c by executing s. Hence, $p(c) \cap \gamma(a) \neq \emptyset$ and we are done.

We now prove the only-if direction. Consider an entry port $e_{a'}$ for a' from a. Since $\gamma(e'_a) \subseteq \gamma(a')$, then clearly $e'_a \Rightarrow a'$. In order to see that $e'_a \Rightarrow \mathrm{SP}(s, a)$, consider a state $c' \in e_{a'}$. Since $p(c') \cap \gamma(a) \neq \emptyset$, there is a state $c \in \gamma(a)$ such that c' is a successor of c. Hence, c' satisfies $\mathrm{SP}(s, a)$, and we are done.

We proceed to exit ports. We first prove the if direction. Let x_a be such that $x_a \Rightarrow a \wedge \mathrm{Pre}(s, a')$. We show that the two conditions for x_a being an exit port for a to a' are satisfied. First, since $x_a \Rightarrow a$, then clearly $\gamma(x_a) \subseteq \gamma(a)$. For the second condition, consider a concrete state $c \in \gamma(x_a)$, Since c satisfies $\mathrm{Pre}(s, a')$, it must have a successor in a', thus $s(c) \cap \gamma(a') \neq \emptyset$ and we are done.

We now prove the only-if direction. Consider an exit port x_a for a to a'. Since $\gamma(x_a) \subseteq \gamma(a)$, then clearly $x_a \Rightarrow a$. In order to see that $x_a \Rightarrow \mathrm{Pre}(s, a')$, assume by way of contradiction that there is $c \in \gamma(x_a)$ such that c does not satisfy $\mathrm{WP}(s, a')$. Then, however, $s(c) \cap \gamma(a') = \emptyset$, contradicting the fact that $s(c) \cap \gamma(a') \neq \emptyset$. $\square$

[6] As noted in [1], if there are abstract states b and b' such that $must^{-*}(a, b)$, $may(b, b')$, and $must^{+*}(b', a')$, we can still conclude that a' is weakly reachable from a. This "one flip trick" is valid also in the reasoning we describe here. For the sake of simplicity, we restrict attention to the closure of either $must^{+}$ or $must^{-}$ transitions.

The lemma suggests that when we glue a_{i-1} to a_i, we proceed with entry port $a_i \wedge \mathrm{SP}(s, a_{i-1})$ for a_i. Then, when we glue state a_i to a_{i+1}, we proceed with exit port $a_i \wedge \mathrm{WP}(s, a_{i+1})$ for a_i.

Example 2. In Example 1, we described an application of our method to the procedure `simple`. The entry and exit ports used in the example (see Figure 5) have been generated automatically using the characterization in Lemma 1. Consider, for example, the states $a_0 : (0 \leq x < n)$ and $a_1 : (n \leq x < 2n)$. Recall that the statement s executed in a_0 is `while x < n do x:=x+1`. The exit port of a_0 is then $a_0 \wedge \mathrm{WP}(s, a_1) = (x = n - 1)$ and the entry port of a_1 is $a_1 \wedge \mathrm{SP}(s, a_0) = (x = n)$.

The ports induced by the Lemma are the maximal ones. Note, however, that the conditions in Theorem 1 and 2 are monotonic with respect to the entry port (the bigger it is, the more likely it is for the conditions to hold), Condition (2) is monotonic and Condition (3) is anti-monotonic with respect to the exit port. Thus, one can always take the maximal entry port (the way we have defined it also guarantees that it is possible to "glue" it to a_{i-1}), start also with a maximal exit port, and search for a subset of the maximal exit port in case Condition (3) does not hold but $must^-(a, a \wedge \neg e_a)$ holds. The search for the subset can use a theorem prover and the characterization of $must^-$ transitions by means of weakest preconditions. Reasoning is dual for Theorem 2.

4.2 Checking the conditions

Once entry and exit ports are established, we proceed to check the conditions in Theorems 1 or 2. In many cases, the program is known to be deterministic, thus the determinism check in Theorem 1 is redundant. Theorem 1, however, is applicable also when the program is nondeterministic, or not known to be deterministic, and we have to check a weaker condition, namely for all $c \in \gamma(a \wedge \neg x_a)$, we have that $\mid s(c) \cap \gamma(a) \mid \leq 1$. In order to automate the check, we use the statement s that is executed in a, and the fact that the successors of a state satisfy $\mathrm{WP}(s, a)$, which can be decomposed for nondeterministic statements. Formally, we have the following.

Lemma 2. *Let $s_1|s_2$ be a nondeterministic statement executed in a, for deterministic statements s_1 and s_2. If there exists $c \in \gamma(a)$ such that $\mid s(c) \cap \gamma(a) \mid > 1$, then the formula $a \wedge \neg x_a \wedge \mathrm{WP}(s_1, a) \wedge \mathrm{WP}(s_2, a)$ is satisfiable.*

Lemma 2 refers to nondeterminism of degree two, and to a statement in which the nondeterminism is external [7].

Similarly, to check the reverse-nondeterminism condition in Theorem 2, we have to find $c \in \gamma(a \wedge \neg e_a)$ such that c is reachable from two states in $\gamma(a)$. If the nondeterministic statement executed in a is $s_1|s_2$ and then there exists $c \in \gamma(a)$ such that $\mid p(c) \cap \gamma(a) \mid > 1$, then the formula $a \wedge \neg e_a \wedge \mathrm{SP}(s_1, a) \wedge \mathrm{SP}(s_2, a)$ is satisfiable.

Checking the local reachability conditions in Theorems 1 and 2 can be done using the characterization of $must^-$ and $must^+$ transitions. Specifically, $must^-(a \wedge \neg x_a, a \wedge$

[7] In order for the second direction of the lemma to hold, one has to check that executing s_1 and s_2 from c results in different states. If this is not the case, then the nondeterminism in a is only syntactic, and one can apply Theorem 1 by disabling one of the nondeterministic choices (see Section 5).

$\neg e_a)$ iff $a \wedge \neg e_a \Rightarrow \mathrm{SP}(s, a \wedge \neg x_a)$ and $must^+(a \wedge \neg x_a, a \wedge \neg e_a)$ iff $a \wedge \neg x_a \Rightarrow \mathrm{WP}(s, a \wedge \neg e_a)$.

Remark 1. An advantage of forward reasoning (Theorem 2) is that the check for $must^+$ transitions involves weakest preconditions, which are often easier to compute than strongest postconditions, which are required for checking $must^-$ transitions. Indeed, for an assignment statement `x:=v`, we have that $\mathrm{WP}(x := v, e) = e[x/v]$ (that is, e with all occurrences of x replaced by v, whereas $\mathrm{SP}(x := v, e) = \exists x'.(e[x/x'] \wedge x = v)$. On the other hand, an advantage of backwards reasoning (Theorem 1) is the fact that checking that a program is deterministic is often easier than checking that it is reverse deterministic, especially in cases the program is known to be deterministic.

4.3 Proceeding without a suggested path

So far, we assumed that weak reachability from a to a' is checked along a path suggested by the user. When the user does not provide such a path, one possible way to proceed is to check all simple paths from a to a'. Since these are paths in the abstract MTS and the cost of each check depends only on the size of the MTS, this is feasible. Alternatively, we can check the path obtained by proceeding in a BFS from a along the MTS. Thus, the path along which we check reachability is $a_0, a_1, \ldots, a_n$, where $a_0 = a$ and a_i is the union of states that are reachable in the MTS from a_{i-1}. We stop at a_n that contains a'. We now try to prove that $must^{-*}(a_0, a_n)$, which implies that a' is weakly reachable from a. We start with $i = n$ and when we come across an iteration i such that $must^-(a_{i-1}, a_i)$ does not hold, we check whether a_{i+1} involves a loop and Theorem 1 is applicable. If this is not the case, we refine a_{i+1}.

5 Making the Method More General

The application of Theorem 1 requires the concrete system to be deterministic with respect to $\gamma(a)$. That is, every concrete state in $\gamma(a)$ should have at most one successor in $\gamma(a)$. As demonstrated in Section 3.2, one way to cope with nondeterminism is to refine a so that, while being nondeterministic, the program is deterministic with respect to $\gamma(a)$. In this section we discuss how generalize our method to handle cases in which the program is nondeterministic and there is no way to refine a efficiently and make it deterministic with respect to a. As an example, consider the procedure `jump_beyond_n` appearing in Figure 9. The figure also depicts the concrete state space. Abstracting it to three abstract states according to the predicates $x = 0, 0 < x < n$, and $x \geq n$ results in the problematic setting of Figure 3, where we cannot conclude that the error state $x \geq n$ is reachable from the initial state $x = 0$. The procedure has a nondeterministic choice (when $x = 1$ mod 1, it can be decreased by either 1 or 3) and there is no way to refine the abstract state $0 < x < n$ so that the conditions of Theorem 1 hold.

Our technique is to generate programs with fewer behaviors, with a hope that we preserve weak reachability and satisfy the conditions of the theorem. The programs we try first are deterministic programs obtained from the original program by disabling some of its nondeterministic choices. In our example, we try to apply Theorem 1 with

```
procedure jump_beyond_n (int n)
  int x:=0
  while (x < n) do case
                    x = 1 mod 2: x:=x-1|x:=x-3;
                    x = 0 mod 4: x:=x+5;
                    x = 2 mod 4: x:=x+1;
  assert false
```

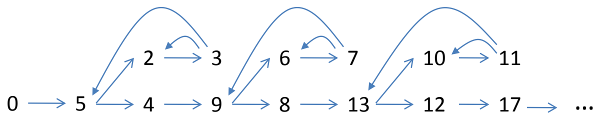

Fig. 9. The `procedure jump_beyond_n`.

respect to the two procedures obtained from `jump_beyond_n` by replacing the statement `x:=x-1|x:=x-3` by `x:=x-1` or `x:=x-3`. As can be seen in the description of the concrete state space, for this example, this would work – going always with `x:=x-1` increments x to go beyond n. Thus, in order to apply Theorem 1, we have to disable the `x:=x-3` branch and refine the abstract state to $2 \leq x < n$ (the state $x = 1$ is unreachable).

Disabling nondeterministic branches works when reachability can be achieved by always taking the same transition. As we discuss below, this is not always possible. A more general approach is to determinize the program by adding predicates that "schedule" the different branches. Thus, a nondeterministic choice $s_1|s_2|\cdots|s_k$ is replaced by `case` $b_1 : s_1; \ldots ; b_k : s_k$, for mutually exclusive predicates $b_1, \ldots , b_k$. The predicates $b_1, \ldots , b_k$ can be automatically generated (for example, proceed in a round-robin fashion among all branches) or can be obtained from the user.

Stationary strategies in weak reachability Reachability in a CTS can be checked along simple paths. On the other hand, since each state in an MTS corresponds to several concrete states, weak reachability may have to traverse the same abstract state several times. Consider a nondeterministic CTS C and its abstract MTS S. Let a and a' be two abstract states in S, and let $\pi = c_0, \ldots , c_n$ be a path in C such that $c_0 \in \gamma(a)$ and $c_n \in \gamma(a')$. We say that π is *stationary* if for all $0 \leq i, j < n$, if $\gamma^{-1}(c_i) = \gamma^{-1}(c_j)$ (that is, c_i and c_j are in the same abstract state), then the same nondeterministic choice was taken in c_i and c_j. We say that there is a stationary strategy to reach a' from a if a' is weakly reachable from a via a stationary path.

In concrete reachability games, if one of the players has a winning strategy, then he also has a stationary strategy, in the sense that the next position of the game depends only on the current position and is independent of the history of the game so far. In our setting, the existence of stationary strategies corresponds to the existence of a deterministic program via which weak reachability can be proven. The procedure `flip` in Figure 10 shows that stationary strategies do not always exist.

While there is no way to disable branches in `flip`, we can apply Theorem 1 to the deterministic procedure obtained from `flip` by replacing the nondeterministic

```
procedure flip
  x:=0; t:=1
  while x < n do
    t:=-t;{x:=x+t|x:=x-t};
  assert false
```

Fig. 10. The procedure `flip`.

statement `x:=x+t|x:=x-t` by the deterministic statement `case t=1:x:=x+t; t=-1: x:=x-t`. Thus, while there is no stationary strategy, there is a simple *two-state* strategy.

References

1. T. Ball. A theory of predicate-complete test coverage and generation. In *3rd International Symposium on Formal Methods for Components and Objects*, 2004.
2. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.
3. T. Ball and O. Kupferman. Better under-approximation of programs by hiding of variables. *Proc. 7th VMCAI*, 2006.
4. J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses *Proc. 34th POPL*, 2007.
5. A.R. Bradley, Z. Manna, and H. Sipma. Linear Ranking with Reachability In Proc. of 17th CAV, LNCS 3576, pages 491–504, 2005.
6. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. ACM PLDI*, pages 415–426, 2006.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, pages 238–252. ACM, 1977.
8. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
9. P. Godefroid and R. Jagadeesan. Automatic abstraction using generalized model checking. In *Proc. 14th CAV*, LNCS 2404, pages 137–150, 2002.
10. D. Kroening and G. Weissenbacher. Counterexamples with loops for predicate abstraction. In *Proc. 18th CAV*, LNCS 4144, pages 152–165, 2006.
11. K.G. Larsen and G.B. Thomsen. A modal process logic. In *Proc. 3th LICS*, 1988.
12. K.G. Larsen and L. XinXin. Equation solving using modal transition systems. In *Proc. 5th LICS*, pages 108–117, 1990.
13. A. Podelski and A. Rybalchenko. Transition invariants. In *Proc. 19th LICS*, pages 32–41, 2004.
14. S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *Proc. TACAS*, LNCS 2988, pages 546–560, 2004.