Theoretical Computer Science 256 (2001) 145–167

Theoretical Computer Science

www.elsevier.com/locate/tcs

Ensuring completeness of symbolic verification methods for infinite-state systems [☆]

Parosh Aziz Abdulla [*], Bengt Jonsson

Department Computer Systems, Uppsala University, P.O.Box 325, 751 05 Uppsala, Sweden

Abstract

Over the last few years there has been an increasing research effort directed towards the automatic verification of infinite state systems. For different classes of such systems, e.g., hybrid automata, data-independent systems, relational automata, Petri nets, and lossy channel systems, this research has resulted in numerous highly nontrivial algorithms. As the interest in this area increases, it will be important to extract common principles that underly these and related results. In this paper, we will present a general model of infinite-state systems, and describe a standard algorithm for reachability analysis of such systems. Our contribution consists in finding conditions under which the algorithm can be fully automated. We perform backward reachability analysis. Using an iterative procedure, we generate successively larger approximations of the set of all states from which a given final state is reachable. We consider classes of systems where these approximations are *well quasi-ordered*, implying that the iterative procedure always terminates. Starting from these general termination conditions, we derive several computations models for which reachability is decidable. Many of these models are extensions of those existing in the literature. Using a well-known reduction from safety properties to reachability properties, we can use our algorithm to decide large classes of safety properties for infinite-state systems. A motivation for our approach is the long-term desire to build general tools for verification of infinite-state systems, which implies that we should employ principles applicable across a rather wide range of such systems. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Infinite-state systems; Model checking; Reachability analysis; Symbolic verification

1. Introduction

In recent years, several approaches to mechanized program verification have been developed. Substantial progress has been made in the development and use of

[☆] Supported in part by the Swedish Board for Industrial and Technical Development (NUTEK) and by the Swedish Research Council for Engineering Sciences (TFR).

[*] Corresponding author.

E-mail address: parosh@docs.uu.se (P.A. Abdulla).

interactive theorem provers, such as PVS [32]. Fully automated techniques have now been developed to the extent that they can routinely handle systems with several millions states (for some applications even several magnitudes more). Partial order techniques [23, 30, 34] and binary decision diagrams (BDDs) [13] have extended the power of these techniques dramatically.

Automated verification of infinite-state systems is becoming practical for significant special cases. Nontrivial verification algorithms have been developed, e.g., for timed automata [2, 1, 15], hybrid automata [24], data-independent systems [28, 36], relational automata [10, 16, 17] Petri nets [26, 27], pushdown processes [14, 33] systems with unbounded communication channels [21, 3, 4], and systems consisting of an unbounded number of identical finite-state processes [22, 29].

A large portion of existing work on verification algorithms for infinite-state systems consider a particular model of an infinite-state system, and exploit its particular properties to develop a special-purpose verification algorithm. However, in order to be able to verify a general class of infinite-state systems, it is necessary to find and exploit common principles that can be uniformly applied for the entire class. In this paper, we will present an approach to verification of infinite-state systems which does not start from a particular model of systems. Rather, we will present a general model together with some general well-known methods for verification. Our contribution consists in finding conditions under which these general methods can be fully automated. In this paper, the criterion for "fully automated" will be decidability, i.e., that the method is guaranteed to terminate. A motivation for our approach is the long-term desire to build general tools for verification of infinite-state systems. In order to be able build a verification tool which can successfully handle a reasonably large class of systems, we must employ principles applicable across a rather wide range of such systems.

We will concentrate our treatment on the verification of reachability properties. For a particular system and two sets of states (a set of initial states and a set of final states) of the system, the reachability problem asks whether some final state can be reached in an execution which starts from an initial state. A typical application of the problem is to check that some undesired situation, such as deadlock, cannot occur in an execution of a system. More general classes of safety properties can be reduced to the reachability problem [35, 23]: the property is represented by a test or observer process, and then one checks whether or not the test process can reach a state where violation of the safety property has been observed.

There are several standard methods for verifying reachability. An approach which is suitable for automation is to systematically search for an execution path from some initial state to some final state. Such a search can be conducted, starting from the set of initial states (forward search) or from the final states (backward search). In this paper, we will consider the backward search method.

A standard technique for backward search, is to attempt to generate the set of all states from which a state in F is reachable, using an iterative procedure. For successively larger j, we compute the set of states from which a state in F can be reached by a sequence of transitions of length less than or equal to j. We obtain the $(j+1)$-st approximation from the jth approximation by adding the *pre-image* of the jth approximation, i.e., the set of states from which a state in the jth approximation can be reached by some transition. If this procedure converges (i.e., the jth and $(j+1)$-st approximations coincide for some j), one checks whether the result intersects the set of initial states of the model. The method just mentioned will become a decision procedure for a specific class of systems, provided that we find a suitable representation of infinite sets of states such that we can

- invent a method for computing pre-images, and
- prove that the iteration always converges.

Note that, of course, the method will not converge for arbitrary classes of infinite-state systems.

A typical approach is to represent infinite sets of states using assertions, which we call *constraints*, in some language. The iterative procedure will then generate a sequence of successively weaker constraints. We will try to enforce convergence, by looking for systems of constraints that disallow infinite sequences of successively weaker constraints.

To present the key ideas of this paper, we have chosen to present a simple example in the next section. In this example, we show how to check a simple reachability property for a simple cache protocol, and at the same time introduce the main concepts of the paper.

Related work: Unifying work for verification of infinite-state systems appears e.g., in the framework of the modal mu-calculus, where Bradfield and Stirling [11] have presented general techniques for verifying temporal properties of infinite-state systems. In [20], Finkel introduces a class of infinite-state systems called *well-structured systems* and describes algorithms to check, e.g., coverability and eventuality properties for this class. In our earlier work [5], we presented similar results for a larger class of infinite-state systems, and showed how they could be applied to particular classes of infinite-state systems. This paper contains part of our earlier results, but presents a more thorough treatment on the verification of reachability properties. A distinguishing feature of this work is, that rather than simply noting that our framework can be extended to some existing models of infinite-state systems from the literature, we try to generate new classes of infinite-state systems in the most general way. In many cases, we derive models that are generalizations of existing models in the literature.

Outline: In the next section, we present the main ideas of the paper through a simple example. In Section 3, we introduce the basic notions of *transitions systems* and *well quasi-orderings*. Well quasi-orderedness is a property which we assume to hold of our constraint systems. We use this property both to find finite representations of constraint systems, and to prove termination of our verification algorithms. In Section 4, we describe a standard algorithm for reachability analysis of finite-state systems. The basic concepts of this algorithm are generalized in Section 5 to enable us to deal with infinite-state systems. In Section 6, we give sufficient conditions to guarantee that a constraint system is closed under a given transition relation. In Section 7, we describe methods to generate new well quasi-orderings from existing ones. In Section 8, we introduce UNITY-like programs and use them for syntactical representation of transition systems. In Sections 9 and 10, we consider examples of constraint systems. We use these examples to give a general scheme for applying the methods developed of Sections 5 and 6 to programs operating on several different data structures, such as natural numbers, sequences, real-time clocks, etc.

2. A simple example

In order to illustrate the ideas more concretely, let us consider a simplified version of the IEEE Futurebus Protocol, which is a transaction-based cache coherence protocol. We use a model of the protocol taken from a tutorial by Rushby [31]. The following description is from Rushby [31].

> The protocol consists of a memory and several processors, each with a local cache, attached to a bus. The caches maintain coherence by snooping all bus transactions and updating their local states appropriately. Memory is organized as *lines*. Each cache can store copies of values of several lines; each copy can be either **shared** or **exclusive**. When a processor needs to read a value from a memory location, it checks whether its line is present in its cache. If it is, the processor obtains the value from the cache; if not (in which case the line is said to be **invalid** for that cache), it issues a **read-shared** transaction on the bus. Any cache that has an **exclusive** copy of that line can respond by sending its value over the bus (and changing the status of its own copy to **shared** or **invalid**), otherwise the memory supplies the value. In either case, the requesting cache will load the line as a **shared** copy when its value comes over the bus (other caches may choose, opportunistically, to do the same). When a processor needs to write a value to a memory location, it checks whether it has an **exclusive** copy of the relevant line in its cache. If not, it first obtains a **shared** copy of the line as described above (unless it already has one), then issues a **read-modified** transaction and changes the status of its copy to **exclusive**. Other caches that have a copy of the line change the status of their copies to **invalid** when they see a **read-modified** transaction. A processor that has an **exclusive** copy of a line can relinquish it by issuing a **write-back** transaction. This causes the cache to write its value back to memory and to mark the line as **invalid**.

A model of the behavior of a single cache line can be formulated as the below program

```
Program IEEE Futurebus
  declare $s, e, i$ : natural numbers
  initially $s = e = 0$
  assign
    read-shared :
      $\langle [] n : 0 \leqslant n \leqslant i :: i \geqslant 1 \rightarrow s, e, i := s + sign(e) + n \ , \ e \ominus 1 \ , \ i - n \rangle$
    []
    read-modified :
          $s \geqslant 1 \rightarrow s, e, i := 0 \ , \ e + 1 \ , \ s + i - 1$
    []
    write-back :
          $e \geqslant 1 \rightarrow s, e, i := s \ , \ e - 1 \ , \ i + 1$
end
```

written in UNITY notation. In this program, the variables s, e, and i denote the number of processors that have a **shared** copy, an **exclusive** copy, and an **invalid** copy of this cache line, respectively. Initially, all copies are invalid, implying that the variables s and e are initially 0. The allowed changes to the program variables are determined by three guarded multiple-assignment statements, one for each type of transaction. The statement corresponding to a read-shared transaction has a parameter n, which may range over integers from 0 to i. The parameter represents the number of processors that change state from invalid to shared. If there is at least one exclusive copy, then the number of exclusive copies decreases by 1. If that copy changes its state to shared, then there will be an extra shared copy. The notation $sign(e)$ denotes 1 if $e \geqslant 1$, and 0 otherwise. The notation $e \ominus 1$ denotes $e - sign(e)$. Thus, the action covers the case that n copies change from invalid to shared, and that an eclusive copy, if one exists, changes to shared. In the case that the exclusive copy changes to invalid, then the statement models a transaction in which $n+1$ copies change from invalid to shared, where $n < i$. The read-modified statement assumes that some cache is in shared; it changes that cache to exclusive, and all other shared to invalid. The write-back statement makes a copy change from exclusive to invalid.

A state of the program is given by the values of s, e and i. Let us therefore represent a state by a triple $\langle \mathtt{s}, \mathtt{e}, \mathtt{i} \rangle$ of natural numbers. Consider the problem of checking the invariant that no two cache copies are simultaneously in state **exclusive**. This amounts to checking that some state $\langle \mathtt{s}, \mathtt{e}, \mathtt{i} \rangle$ with $\mathtt{e} \geqslant 2$ is not reachable from any initial state of the program (recall that an initial state satisfies $s = e = 0$). This problem can be solved, using backward reachability analysis, as follows.

Let F be the set $\{\langle \mathtt{s}, \mathtt{e}, \mathtt{i} \rangle : \mathtt{e} \geqslant 2\}$. We use predicates which we call *constraints* to characterize sets of states. In this example, we use constraints of the form $\phi_{a,b,c}$ denoting the set of states $[\![\phi_{a,b,c}]\!] = \{\langle \mathtt{s}, \mathtt{e}, \mathtt{i} \rangle : \mathtt{s} \geqslant a \,\&\, \mathtt{e} \geqslant b \,\&\, \mathtt{i} \geqslant c\}$. We also use finite sets of constraints, where a set Φ of constraints denotes the set $[\![\Phi]\!] = \bigcup_{\phi \in \Phi} [\![\phi]\!]$ of states. We perform a backward reachability analysis from F. For successively larger j, we compute Φ_j, such that $[\![\Phi_j]\!]$ is the set of states from which a state in F can be

j	Φ_j	rule	new constraints generated	covered / added
0	$\phi_{0,2,0}$	**r-s**	$\phi_{0,3,1}$	covered by $\phi_{0,2,0}$
		r-m	$\phi_{1,1,0}$	added to Φ_1
		w-b	$\phi_{0,3,0}$	covered by $\phi_{0,2,0}$
1	$\phi_{0,2,0},\ \phi_{1,1,0}$	**r-s**	$\phi_{0,3,1},\ \phi_{0,2,1}$	covered by $\phi_{0,2,0}$
		r-m	$\phi_{1,1,0}$	already in Φ_1
		w-b	$\phi_{0,3,0},\ \phi_{1,2,0}$	covered by $\phi_{0,2,0}$

Fig. 1. Table representing verification of the simplified example.

reached by a sequence of at most j executions of statements in the program. We start by $\Phi_0 = \{\phi_{0,2,0}\}$, i.e. $[\![\Phi_0]\!] = F$.

A state in $[\![\Phi_0]\!]$ can be reached either by executing **read-shared** from a state in $[\![\Phi_{0,3,1}]\!]$, by executing **read-modified** from a state in $[\![\phi_{1,1,0}]\!]$, or by executing **write-back** from a state in $[\![\Phi_{0,3,0}]\!]$. Thus $\Phi_1 = \Phi_0 \cup \{\phi_{0,3,1}, \phi_{1,1,0}, \phi_{0,3,0}\} = \{\phi_{0,2,0}, \phi_{0,3,1}, \phi_{1,1,0}, \phi_{0,3,0}\}$. Observe that the constraints in Φ_1 are either included in Φ_0, or are obtained by taking the preimage with respect to a program statement. At this point, we note that $\phi_{0,3,1}$ and $\phi_{0,3,0}$ are both *covered* by $\phi_{0,2,0}$, i.e. $[\![\Phi_{0,3,1}]\!] \subseteq [\![\Phi_{0,2,0}]\!]$ and $[\![\Phi_{0,3,0}]\!] \subseteq [\![\Phi_{0,2,0}]\!]$. Therefore, it is meaningless to add $\phi_{0,3,1}$ and $\phi_{0,3,0}$ to Φ_1, since $\phi_{0,2,0}$ is already in Φ_0. Thus $\Phi_1 = \{\phi_{0,2,0}, \phi_{1,1,0}\}$.

A state in Φ_1 can be reached either by executing **read-shared** from a state in $[\![\{\phi_{0,3,1}, \phi_{0,2,1}\}]\!]$, by executing **read-modified** from a state in $[\![\phi_{1,1,0}]\!]$, or by executing **write-back** from a state in $[\![\{\phi_{0,3,0}, \phi_{1,2,0}\}]\!]$. Thus, $\Phi_2 = \{\phi_{0,2,0}, \phi_{1,1,0}, \phi_{0,3,1}, \phi_{0,2,1}, \phi_{1,1,0}, \phi_{0,3,0}, \phi_{1,2,0}\}$, which by removing redundant constraints can be written as $\{\phi_{0,2,0}, \phi_{1,1,0}\}$. As a result, we obtain $\Phi_2 = \{\phi_{0,2,0}, \phi_{1,1,0}\} = \Phi_1$.

A simulation of the verification is described in Fig. 1 as follows. For each j, we describe the constraints in Φ_j. We also show the constraints generated by applying the statements **r-s** (read-shared), **r-s** (read-modified), and **r-s** (write-back) backwards to the constraints in Φ_j. we describe for each constraint, whether it is covered by an already generated constraint, and hence discarded, or added to Φ_{j+1}.

Notice that $\Phi_2 = \Phi_1$, implying that $\Phi_j = \Phi_1$ for all $j \geqslant 1$. We conclude that F is reachable only from states in $[\![\Phi_1]\!]$. Since $\langle 0, 0, k\rangle \notin [\![\Phi_1]\!]$ for any $k \geqslant 0$ (an easy check), we conclude that F is not reachable from any initial state of the program.

Let us examine the above procedure in order to find out what general principles emerge. We are computing a fixpoint by an iterative procedure. Each approximation Φ_j can be described as a finite set of constraints. For a statement s, let $pre(s, \Phi)$ denote the set of states from which a state in Φ is reachable through one application of s. The following properties are essential for our algorithm.

(1) For each finite set Φ of constraints we should be able to compute $pre(s, \Phi)$ as a finite set of constraints, i.e., the constraint system should be closed under applications of the three program statements. In Section 6, we show that this property is equivalent to the statements of the program being *monotone*, in the sense that if a

statement transforms the state $\langle \mathtt{s}, \mathtt{e}, \mathtt{i} \rangle$ into $\langle \mathtt{s}', \mathtt{e}', \mathtt{i}' \rangle$ then any state $\langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$ with $\mathtt{s} \leqslant \mathtt{a}$, $\mathtt{e} \leqslant \mathtt{b}$, and $\mathtt{i} \leqslant \mathtt{c}$ is transformed into a state $\langle \mathtt{a}', \mathtt{b}', \mathtt{c}' \rangle$ with $\mathtt{s}' \leqslant \mathtt{a}'$, $\mathtt{e}' \leqslant \mathtt{b}'$, and $\mathtt{i}' \leqslant \mathtt{c}'$. We can then conclude that if F can be represented by a finite set of constraints, then all Φ_j can be computed as finite sets of constraints.

(2) Several times, we were able to discard constraints $\phi_{\mathtt{s},\mathtt{e},\mathtt{i}}$ since they *entailed* (were covered by) some other constraint $\phi_{\mathtt{a},\mathtt{b},\mathtt{c}}$. More precisely, $\phi_{\mathtt{s},\mathtt{e},\mathtt{i}}$ *entails* $\phi_{\mathtt{a},\mathtt{b},\mathtt{c}}$ if $\mathtt{a} \leqslant \mathtt{s}$, $\mathtt{b} \leqslant \mathtt{e}$, and $\mathtt{i} \leqslant \mathtt{c}$, (implying $[\![\phi_{\mathtt{s},\mathtt{e},\mathtt{i}}]\!] \subseteq [\![\phi_{\mathtt{a},\mathtt{b},\mathtt{c}}]\!]$). In this way, the growth of the Φ_j's was slowed down, so that eventually convergence was reached. In this example, it turns out that entailment among constraints of form $\phi_{\mathtt{s},\mathtt{e},\mathtt{i}}$ has an important property, which guarantees convergence; namely, entailment is a *well quasi-ordering* (Section 3.2 and Section 7). Roughly, this means that in any sequence of added constraints of form $\phi_{\mathtt{s},\mathtt{e},\mathtt{i}}$, we will eventually reach a situation where any subsequent constraint will be redundant, and hence discarded. Intuitively, we could note this effect in the example: "fresh" constraints $\phi_{\mathtt{s},\mathtt{e},\mathtt{i}}$ had to introduce a component (a or b) which was less than the already produced constraints. Clearly, we cannot produce an infinite sequence of constraints that are "fresh" in this sense.

This general observation can be used to conclude that, e.g., for programs over natural numbers, containing only monotone statements, the problem of checking reachability of a constraint of the above form is decidable. We observe that Petri nets are a special case of such programs, and conclude that the coverability problem for Petri nets is decidable. These ideas can be generalized (Sections 7, 9, and 10) to much richer classes of programs, e.g., to programs that operate over compound data structures such as sequences, sets, and multi-sets. The only thing that must be done is to generalize the ordering to the data domain in question, and to consider programs whose statements are monotone with respect to this ordering. If the ordering is still a well quasi-ordering, this guarantees decidability of the reachability problem.

3. Preliminaries

We introduce the notions of transition systems and well quasi-orderings.

3.1. Transition systems

We present the basic definitions for transition systems.

Definition 3.1. A *transition system* $\mathscr{T}$ is a pair $\langle \Sigma, \longrightarrow \rangle$, where Σ is a set of *states*, and $\longrightarrow \subseteq \Sigma \times \Sigma$ is a *transition relation.*

According to Definition 3.1, a transition relation denotes a set of pairs of states. Each such pair is called a *transition.* We use $\sigma_1 \longrightarrow \sigma_2$ to denote that $(\sigma_1, \sigma_2) \in \longrightarrow$. Let $\longrightarrow^*$ denote the transitive and reflexive closure of $\longrightarrow$. We say that a state σ_2 is *reachable* from a state σ_1 if $\sigma_1 \rightarrow^* \sigma_2$. For two sets *Init* and F of states, we say that F is *reachable* from *Init* if some state σ_F in F is reachable from a state σ_I in *Init*.

For a set $S \subseteq \Sigma$ of states and a binary relation $\rightarrow$ on Σ, we use $pre(\rightarrow, S)$ to denote the set $\{\sigma : \exists \sigma' \in S : \sigma \rightarrow \sigma'\}$ of states from which a state in S can be reached via a transition in $\rightarrow$.

The reachability problem is defined as follows.

Instance: A transition system $\langle \Sigma, \rightarrow \rangle$ and two sets of states *Init* and F.

Question: Is F reachable from *Init*?

The reachability problem is central to automated verification, since it can be used to verify a large class of safety properties of transition systems [35, 23].

3.2. Well quasi-orderings

We introduce the notion of *well quasi-orderings*.

A *quasi-order*[1] $\preceq$ on A is a binary relation over A which is reflexive and transitive. A set $I \subseteq A$ is said to be an *ideal* (*with respect to* $\preceq$) if it is the case that $a \in I$ and $a \preceq b$ imply $b \in I$. The *ideal generated by* $a \in A$, denoted $id(\preceq, a)$, is defined to be the set $\{b : a \preceq b\}$. We say that $B \subseteq A$ is a *minor set* of A, if (i) for all $a \in A$ there exists $b \in B$ such that $b \preceq a$, and (ii) $a, b \in B$ and $a \preceq b$ imply $a = b$.

Definition 3.2. A quasi-order $\preceq$ on a set A is a *well quasi-ordering* (wqo) if in each infinite sequence $a_0\ a_1\ a_2\ a_3 \cdots$ of elements in A, there are indices $i < j$ such that $a_i \preceq a_j$.

Intuively, a well quasi-ordering has the property that each infinite sequence of elements contains a pair of ordered elements. We observe that each well quasi-ordering is well-founded. However the converse is not true. For example the prefix relation among strings over a given alphabet is well-founded but not a well quasi-ordering.

Example 3.3. An example of a well quasi-ordering is the identity relation on any finite set. Another example is the "less-than-or-equal" relation $\leqslant$ on the set $\mathcal{N}$ of natural numbers. However, the relation $\leqslant$ is *not* a well quasi-ordering on the set of integers, nor on the set of nonnegative rational numbers.

Proposition 3.4. *For a set A and a well quasi-ordering* $\preceq$ *on A, there exists at least one finite minor set of A.*

Proof. Suppose that no finite minor set of A exists. We show that $\preceq$ is not a well quasi-ordering. We define the infinite sequence $a_0, a_1, a_2, \ldots$ of elements in A as follows. Let a_0 be any arbitrary element in A. We choose a_{i+1} such that $a_j \not\preceq a_{i+1}$ for each $j : 1 \leqslant j \leqslant i$. The element a_{i+1} exists, since otherwise $\{a_0, a_1, \ldots, a_i\}$ would be a minor

[1] Frequently, the term *preorder* is used instead.

```
**Procedure** *Reachable*1
**Input**
    $\mathscr{T} = \langle \Sigma, \rightarrow \rangle$: transition system
    *Init*: set of initial states
    $F$: set of final states finite set of constraints
**Output** are there $\sigma \in Init$ and $\sigma' \in [\![\Phi_F]\!]$ such that $\sigma \rightarrow^* \sigma'$?

**var** $V$: set of states
**begin**
    $V := F$;
    **while** $pre(\rightarrow, V) \not\subseteq V$ **do** $V := V \cup pre(\rightarrow, V)$ **od**;
        **if** $Init \cap V = \emptyset$ **then return** unreachable **else return** reachable
**end**
```

Fig. 2. Algorithm for deciding reachability.

set of A, contradicting the assumption that no such sets exist. It is clear that the sequence $a_0, a_1, a_2, \ldots$ violates the well quasi-orderedness property. □

4. Finite-state verification

In this section we review a standard approach (*Reachable*1, Fig. 2) to solving the reachability problem in the case that the set Σ of states of the transition system $\langle \Sigma, \rightarrow \rangle$ is finite. The reachability problem can be solved by a systematically enumerating all states σ from which some state $\sigma_F \in F$ is reachable (i.e., computing $pre(\rightarrow^*, F)$), and then checking whether some state in *Init* is contained in this set. In Fig. 2 we give a naive presentation of this idea.

The above procedure is guaranteed to terminate since the value of the variable V increases at each iteration of the while-loop, but is bounded by the finite set Σ.

Let us refine *Reachable*1 into a more pragmatic version (*Reachable*2, Fig. 3) which does not manipulate entire sets, but rather computes preimages for individual states. This is the way that reachability analysis is normally implemented. In this refinement, the set V (in *Reachable*1) is represented (in *Reachable*2) as the union of two sets: a set V of states whose predecessors have already been generated, and a set W of states whose predecessors have not yet been generated. The analysis works by repeatedly choosing from the already generated states in W one which is so far "unexplored" and adding its predecessors to the set of generated states. If during the exploration some state in *Init* is generated, then F is reachable from *Init*, otherwise F is not reachable from *Init*.

A description of the algorithm is given in Fig. 3. In the algorithm, the set W is the set of visited but still unexplored states, whereas V is the set of visited and explored states.

```
Procedure Reachable2
Input
    $\mathscr{T} = \langle \Sigma, \rightarrow \rangle$: transition system
    Init: set of initial states
    $F$: set of final states finite set of constraints
Output are there $\sigma \in$ Init and $\sigma' \in [\![\Phi_F]\!]$ such that $\sigma \rightarrow^* \sigma'$?

var $W, V$: sets of states
begin
    $W, V := F, \emptyset$;
    while $W \neq \emptyset$ do
        choose $\sigma \in W$;
        $W := W \backslash \{\sigma\}$;
        if $\sigma \in$ Init then return reachable
        else
            if $\sigma \notin V$
            then
                $V := V \cup \{\sigma\}$;
                $W := W \cup pre(\rightarrow, \{\sigma\})$
    od;
    return unreachable
end
```

Fig. 3. Algorithm for deciding reachability.

One step in the exploration consists of choosing an unexplored state σ. If $\sigma \in \mathit{Init}$ then we have found a path "backwards" from F to *Init*. Otherwise we check whether σ is already explored. If not, we add the predecessors of σ to the set of unexplored states, and move σ to the set of explored states. The algorithm in Fig. 3 trivially terminates because there are only a finite number of states that can be added to V.

5. Symbolic verification

In this section drop the assumption that the program is finite-state. We investigate a "symbolic" generalization of the algorithm in Fig. 3. Instead of letting the reachability algorithm manipulate individual states, we let it manipulate sets of states, represented by predicates which we call *constraints*. Each constraint may characterize a finite or an infinite set of states. In our generalization, we will require that F is represented as a finite union Φ_F of constraints, while *Init* is represented by the negation of a finite union Φ_{Init} of constraints. The sets V and W in the program will also be finite unions of constraints. In each step of the procedure, we consider an "unexplored" constraint and generate its predecessors, again represented by a finite union of constraints. Note that we here make the nontrivial assumption that we can indeed represent the predecessors of a constraint as a finite union of constraints. If the search generates some constraint representing a set of states that has a nonempty intersection with *Init*, then F is reachable from *Init*, otherwise F is not reachable from *Init*.

The choice of constraints in the reachability analysis may depend on the program, so let us define the notion of constraint system relative to a given transition system.

Definition 5.1. Let $\mathscr{T} = \langle \Sigma, \longrightarrow \rangle$ be a transition system. A *constraint system* $\mathscr{C}$ for $\langle \Sigma, \longrightarrow \rangle$ is a set of objects, called *constraints*, where to each ϕ is assigned a denotation $[\![\phi]\!] \subseteq \Sigma$. We define a quasi-order $\sqsubseteq$ on $\mathscr{C}$, where $\phi \sqsubseteq \phi'$ if $[\![\phi']\!] \subseteq [\![\phi]\!]$. Intuitively, $\phi \sqsubseteq \phi'$ denotes that ϕ' "entails" ϕ, or that ϕ' is "stronger than" ϕ. A set Φ of constraints denotes the union of the denotations of its elements, i.e., $[\![\Phi]\!] = \bigcup_{\phi \in \Phi} [\![\phi]\!]$. The definition of $\sqsubseteq$ is extended to sets of constraints, so that $\Phi \sqsubseteq \Phi'$ if $[\![\Phi']\!] \subseteq [\![\Phi]\!]$. In the sequel we assume that $\sqsubseteq$ is computable among finite sets of constraints.

Example 5.2. In the example of Section 2, we have e.g. $\phi_{0,2,0} \sqsubseteq \phi_{0,3,1}$.

In the context of finite transition systems, one of the most well-known constraint systems is *Binary decision diagrams* (BDDs) [12]. Each BDD is a constraint representing a propositional logical formula, and thus characterizes a subset of Σ. In this paper, we consider constraint systems which allow us to analyze infinite state spaces.

Definition 5.3. For a transition system $\mathscr{T} = \langle \Sigma, \longrightarrow \rangle$ and a constraint system $\mathscr{C}$, we say that $\mathscr{C}$ is (*effectively*) *closed* with respect to $\mathscr{T}$, if for each $\phi \in \mathscr{C}$, we can compute a finite set $\Phi \subseteq \mathscr{C}$ such that $pre(\longrightarrow, [\![\phi]\!]) = [\![\Phi]\!]$.

Definition 5.4. We say that a constraint system $\mathscr{C}$ is *well quasi-ordered* if the relation $\sqsubseteq$ is a well quasi-ordering on the elements of $\mathscr{C}$.

We can now generalize the procedure in Fig. 3 in a straight-forward manner, obtaining the procedure in Fig. 4. The procedure assumes that we have selected a constraint system which is effectively closed with respect to the transition system. The procedure *Reachable*3 in Fig. 4 works in the same way as *Reachable*2 in Fig. 3, with individual states being replaced by constraints. A more natural version of the insertion test is the condition $[\![\phi]\!] \not\subseteq [\![V]\!]$. This test may be expensive to perform, in the case where V contains a large number of constraints. In many practical situations it is cheaper to have the weaker insertion test shown in Fig. 4.

The following theorem gives sufficient conditions for decidability of the reachability algorithm.

Theorem 5.5. *For a well quasi-ordered constraint system $\mathscr{C}$, and a transition system $\mathscr{T}$, if $\mathscr{C}$ is effectively closed with respect $\mathscr{T}$ then the reachability problem is decidable.*

```
Procedure Reachable3($\langle\Sigma,\rightarrow\rangle$, $Init$, $F$)
Input
  $\mathscr{T} = \langle\Sigma,\rightarrow\rangle$: transition system
  $Init$: set of states, $Init = \neg[\![\Phi_{Init}]\!]$ for some finite set $\Phi_{Init}$ of constraints
  $\Phi_F$: finite set of constraints
Output are there $\sigma \in Init$ and $\sigma' \in [\![\Phi_F]\!]$ such that $\sigma \rightarrow^* \sigma'$?

var $W, V$: sets of constraints
begin
  $W, V := \Phi_F, \emptyset$;
  while $W \neq \emptyset$ do
    choose $\phi \in W$;
    $W := W \backslash \{\phi\}$;
    if $[\![\phi]\!] \cap Init \neq \emptyset$ then return reachable
    else
      if $\nexists \phi' \in V : \phi' \sqsubseteq \phi$                (* insertion test *)
      then
        $V := V \cup \{\phi\}$;
        $W := W \cup \Phi$ where $[\![\Phi]\!] = pre(\rightarrow, [\![\phi]\!])$;
  od
  return unreachable
end
```

Fig. 4. Symbolic reachability algorithm.

Proof. Since $\mathscr{C}$ is closed with respect to $\mathscr{T}$, and $\sqsubseteq$ is computable it follows that each iteration of the loop can be performed effectively. Observe that the test $[\![\phi]\!] \cap Init \neq \emptyset$ is equivalent to the negation of $\Phi_{Init} \sqsubseteq \{\phi\}$, which can be checked since Φ_{Init} is finite and $\sqsubseteq$ is assumed (Definition 5.1) to be computable among finite sets of constraints.

Termination follows from the following argument. Consider the sequence $\phi_1\phi_2\phi_3\ldots$ of constraints that are added to V. Due to the test $\nexists\phi' \in V : \phi' \sqsubseteq \phi$, we have $\phi_i \not\sqsubseteq \phi_j$ for all $i < j$. Since $\sqsubseteq$ is a well quasi-ordering, the sequence cannot be infinite, and hence each execution the algorithm will terminate after a finite number of iterations of the main loop. □

From Theorem 5.5 we conclude that two main challenges in designing Reachability3 for a certain class of systems, are

(1) *Closedness*: To prove the closedness of the constraint system with respect the transition system. In Section 6, we describe sufficient conditions for achieving closedness.
(2) *Termination*: To investigate conditions under which termination of the algorithm can be guaranteed. In this paper we achieve termination through the assumption

that our constraint systems are well quasi-ordered. In Section 7, we present a method to generate new well quasi-orderings from existing ones.

6. Closedness

We introduce the notion of *monotonicity* for transition systems. We show (Theorems 6.2 and 6.8) that the closedness of a constraint system ϕ with respect to a transition system $\mathscr{T}$, can be characterized as whether $\mathscr{T}$ is *monotone* with respect to a preorder derived from $\mathscr{C}$ in a natural way.

Definition 6.1. Let $\mathscr{T} = \langle \Sigma, \rightarrow \rangle$ be a transition system, with a preorder $\preceq$ defined on Σ. We say that $\mathscr{T}$ is *monotone with respect to* $\preceq$ if, for any states σ_1, σ_2, and σ_3, with $\sigma_1 \preceq \sigma_2$ and $\sigma_1 \rightarrow \sigma_3$, there exists a state σ_4 such that $\sigma_3 \preceq \sigma_4$ and $\sigma_2 \rightarrow \sigma_4$.

Theorem 6.2. *For a transition system $\mathscr{T} = \langle \Sigma, \rightarrow \rangle$, which is monotone with respect to a preorder $\preceq$ on $\mathscr{T}$, if $I \subseteq \Sigma$ is an ideal, then $pre(\rightarrow, I)$ is also an ideal.*

Proof. Suppose that $\sigma_1 \in pre(\rightarrow, I)$ and $\sigma_1 \preceq \sigma_2$. We show that $\sigma_2 \in pre(\rightarrow, I)$. We know that there is $\sigma_3 \in I$ such that $\sigma_1 \rightarrow \sigma_3$. By monotonicity it follows that there is a state σ_4 such that $\sigma_3 \preceq \sigma_4$ and $\sigma_2 \rightarrow \sigma_4$. Since I is an ideal, we get $\sigma_4 \in I$, and hence $\sigma_2 \in pre(\rightarrow, I)$. $\square$

Definition 6.3. Let $\mathscr{C}$ be a constraint system. We define a preorder $\preceq_{\mathscr{C}}$ on Σ, such that $\sigma_1 \preceq_{\mathscr{C}} \sigma_2$ if and only $\sigma_1 \in [\![\phi]\!]$ implies $\sigma_2 \in [\![\phi]\!]$, for each constraint $\phi \in \mathscr{C}$.

Proposition 6.4. *For a constraint system $\mathscr{C}$, each constraint $\phi \in \mathscr{C}$ is an ideal with respect to $\preceq_{\mathscr{C}}$.*

Proof. The proof follows immediately from the definitions. $\square$

Definition 6.5. A constraint system $\mathscr{C}$ is said to be *disjunctive* if for each $\phi_1, \phi_2 \in \mathscr{C}$, there is a (possibly infinite) set $\Psi \subseteq \mathscr{C}$ such that $[\![\phi_1]\!] \cap [\![\phi_2]\!] = \bigcup_{\phi \in \Psi} [\![\phi]\!]$.

Proposition 6.6. *For a disjunctive constraint system $\mathscr{C}$, and a transition system $\mathscr{T} = \langle \Sigma, \rightarrow \rangle$ if $I \subseteq \Sigma$ is an ideal with respect to $\preceq_{\mathscr{C}}$, then there is a (possibly infinite) set $\Psi \subseteq \mathscr{C}$ such that $I = \bigcup_{\phi \in \Psi} [\![\phi]\!]$.*

Proof. We define $\Psi = \bigcup_{[\![\phi]\!] \subseteq I} \phi$. It is obvious that $[\![\Psi]\!] \subseteq I$. We show that $I \subseteq [\![\Psi]\!]$. Suppose that $\sigma \in I$. We prove that $\sigma \in [\![\Psi]\!]$.

From the definitions we know that $id(\preceq_{\mathscr{C}}, \sigma) = \bigcap_{\sigma \in [\![\phi]\!]} [\![\phi]\!]$. Since $\sigma \in I$ and I is an ideal, it follows that $id(\preceq_{\mathscr{C}}, \sigma) \subseteq I$, i.e. $\bigcap_{\sigma \in [\![\phi]\!]} [\![\phi]\!] \subseteq I$. We know that $\mathscr{C}$ is disjunctive, and hence there exists a set $\Psi' \subseteq \mathscr{C}$ such that $\bigcap_{\sigma \in \phi} [\![\phi]\!] = \bigcup_{\phi \in \Psi'} [\![\phi]\!]$. From the above it follows that $\sigma \in id(\preceq_{\mathscr{C}}, \sigma) = \bigcup_{\phi \in \Psi'} [\![\phi]\!]$ and hence there is a $\phi \in \Psi'$ such that $\sigma \in [\![\phi]\!]$.

It also follows that $\bigcup_{\phi\in\Psi'} [\![\phi]\!] \subseteq I$ and hence $[\![\phi]\!] \subseteq I$ for each $\phi \in \Psi'$. This implies that there is a constraint ϕ such that $[\![\phi]\!] \subseteq I$ and $\sigma \in [\![\phi]\!]$. By definition of Ψ it follows that $\sigma \in [\![\Psi]\!]$. $\square$

Corollary 6.7. *For a disjunctive and well quasi-ordered constraint system $\mathscr{C}$, and a transition system $\mathscr{T} = \langle\Sigma, \longrightarrow\rangle$, if $I \subseteq \Sigma$ is an ideal with respect to $\preceq_{\mathscr{C}}$, then there is a finite set $\Phi \subseteq \mathscr{C}$ such that $I = \bigcup_{\phi\in\Phi} [\![\phi]\!]$.*

Proof. From Proposition 6.6, we know that there is a set $\Psi \subseteq \mathscr{C}$ such that $I = \bigcup_{\phi\in\Psi} [\![\phi]\!]$. We take Φ to be any minor set of Ψ. The existence of Φ follows from Proposition 3.4 and the well quasi-orderedness of $\mathscr{C}$. $\square$

Observe that the properties described in Proposition 6.6 and Corollary 6.7 are not dependent on the transition relation $\longrightarrow$.

Theorem 6.8. *A disjunctive and well quasi-ordered constraint system $\mathscr{C}$ is closed with respect to a transition system $\mathscr{T}$, if and only if $\mathscr{T}$ is monotone with respect to $\preceq_{\mathscr{C}}$.*

Proof. (if): Suppose that $\mathscr{T}$ is monotone with respect to $\preceq_{\mathscr{C}}$. Let $\phi \in \mathscr{C}$. We show that there is a finite set $\Phi \subseteq \mathscr{C}$ such that $pre(\longrightarrow, [\![\phi]\!]) = [\![\Phi]\!]$. By Proposition 6.4 it follows that ϕ is an ideal with respect to $\preceq_{\mathscr{C}}$. By Theorem 6.2 we know that $pre(\longrightarrow, [\![\phi]\!])$ is an ideal. The result follows Corollary 6.7.

(only if): Suppose that $\sigma_1 \preceq \sigma_2$ and $\sigma_1 \longrightarrow \sigma_3$. We show that there exists a state σ_4 such that $\sigma_3 \preceq \sigma_4$ and $\sigma_2 \longrightarrow \sigma_4$. By Corollary 6.7 it follows that there is a finite set $\Phi \subseteq \mathscr{C}$ such that $[\![\Phi]\!] = id(\preceq_{\mathscr{C}}, \sigma_3)$. Since $\mathscr{C}$ is closed with respect to $\mathscr{T}$, it follows that there is a finite $\Phi' \subseteq \mathscr{C}$ such that $[\![\Phi']\!] = pre(\longrightarrow, [\![\Phi]\!])$. We know that there is a $\phi \in \Phi'$ such that $\sigma_1 \in \phi$. From the fact that $\sigma_1 \preceq_{\mathscr{C}} \sigma_2$ we get $\sigma_2 \in [\![\phi]\!]$, and hence $\sigma_2 \in pre(\longrightarrow, [\![\Phi]\!])$. This implies that there is a state σ_4 such that $\sigma_2 \longrightarrow \sigma_4$ and $\sigma_4 \in [\![\Phi]\!] = id(\preceq_{\mathscr{C}}, \sigma_3)$. Consequently $\sigma_3 \preceq_{\mathscr{C}} \sigma_4$. $\square$

7. Building well quasi-orders

We will describe techniques for generating well quasi-ordered constraint systems on compound data structures such as sequences, bags, sets, etc. To do that we restate two standard lemmas, which allow us to lift well quasi-orderings from elements to bags and to sequences. Let A^* denote the set of finite strings over A, and let A^B denote the set of finite bags over A. For a natural number n, let $\hat{n}$ denote the set $\{1, \ldots, n\}$. An element of A^* and of A^B can be represented as a mapping $w\colon \widehat{|w|} \mapsto A$ where $|w|$ is the size of the bag or the length of the sequence. Given a quasi-order $\preceq$ on a set A, define the quasi-order $\preceq^*$ on A^* by letting $w \preceq^* w'$ if and only if there is a monotone[2] injection

[2] Meaning that $h(j_1) \leqslant h(j_2)$ if and only if $j_1 \leqslant j_2$.

$h:\widehat{|w|} \mapsto \widehat{|w'|}$ such that $w(j) \preceq w'(h(j))$ for $1 \leqslant j \leqslant |w|$. Define the quasi-order $\preceq^B$ on bags of A by $w \preceq^* w'$ if and only if there is a (not necessarily monotone) injection $h:\widehat{|w|} \mapsto \widehat{|w'|}$ such that $w(j) \preceq w'(h(j))$ for $1 \leqslant j \leqslant |w|$.

Lemma 7.1. *If $\preceq$ is a wqo on A, then $\preceq^*$ is a wqo on A^* and $\preceq^B$ is a wqo on A^B.*

Proof. The proof can be found in [25]. □

In the following, we give some examples of applications of Lemma 7.1.

Example 7.2. (1) The equality relation on a finite set A is trivially a well quasi-ordering.

(2) The *less-than-equal* relation $\leqslant$ on the $\mathcal{N}$ of natural numbers is a well quasi-ordering.

(3) Consider the set A^* of finite strings over a finite set A. Let $\preceq$ denote the *substring relation* on A^*, i.e. $w_1 \preceq w_2$ if w_1 is a (not necessarily contiguous) substring of w_2. According to Lemma 7.1 and 1, the relation is a well quasi-ordering (Higman's lemma [25]).

(4) Consider the *subbag relation* $\preceq$ on the set $\mathcal{N}^B$ of bags of natural numbers. The relation is defined by $b_1 \preceq b_2$ if there is an injection from the elements of b_1 to the elements of b_2, such that $x \leqslant h(x)$, for each element x in b_1. According Lemmas 7.1 and 7.2, the relation is a well quasi-ordering.

(5) Consider the set of vectors of natural numbers of length n for some $n \geqslant 1$. Define a preorder $\preceq$ on the set such that $\langle x_1,\dots,x_n\rangle \preceq \langle y_1,\dots,y_n\rangle$ if $x_i \leqslant y_i$ for each i: $1 \leqslant i \leqslant n$. According Lemmas 7.1 and 7.2, the relation is a well quasi-ordering (Dickson's lemma [19]).

8. Programs

In this section, we present *programs* and use them as a simple notation for syntactical representation of transition systems. The notation is very similar to the action systems by Back and Kurki-Suonio [9] and to UNITY by Chandy and Misra [15].

We assume a set of *domains*. We assume an *assertion language* containing typed variables, functions, constants and predicates, for forming expressions and assertions. We use $dom(x)$ to denote the domain of the variable x, and more generally we use $dom(e)$ to denote the domain of the expression e. An assertion ψ with free variables $x_1,\dots,x_n$ denotes a subset $[\![\psi]\!]$ of $\mathscr{D}_1 \times \cdots \times \mathscr{D}_n$, where $\mathscr{D}_i = dom(x_i)$, in the usual way. We will use an overbar notation for tuples and write, e.g., $\bar{x}$ instead of $x_1,\dots,x_n$.

Definition 8.1. A *program P* consists of
- a set V of (typed) *program variables*,
- a finite set of *actions*. Each action is a guarded command of form

$$g \rightarrow v_1,\dots,v_n := e_1,\dots,e_n,$$

where the guard g is an assertion, $v_1,\ldots,v_n$ are program variables, and $e_1,\ldots,e_n$ are expressions.[3] The guard g and the expressions $e_1,\ldots,e_n$ may contain free variables, some of which (but not necessarily all) are program variables. The free variables in the action which are not in V are called the *parameters* of the action.

A *variant* of α is an action of form $g[\bar{d}/\bar{x}] \rightarrow \bar{v} := \bar{e}[\bar{d}/\bar{x}]$ obtained by replacing the parameters $x_1,\ldots,x_m$ by values $d_1,\ldots,d_m$ such that $d_i \in dom(x_i)$ for $i = 1,\ldots,m$.

Definition 8.2. A program P denotes a transition system $[\![P]\!] = \langle \Sigma, \rightarrow \rangle$, where

Σ is the set of mappings from program variables to values, such each $v \in V$ is mapped to a value $\sigma(v) \in dom(v)$. A state[4] can be extended to a mapping from expressions and assertions in the natural way.

$\rightarrow$ is the set of pairs (σ, σ') such that for some variant $g \rightarrow \bar{v} := \bar{e}$ of some action of P, we have

- $\sigma(g) = true$, i.e., g is true in σ, and
- $\sigma' = \sigma[\sigma(\bar{e})/\bar{v}]$ i.e., σ' is obtained from σ by performing the assignment $\bar{v} := \bar{e}$.

Let us now assume that each domain $\mathscr{D}_i$ is equipped with a preorder $\preceq$.

Definition 8.3. Let $g(\bar{v}, \bar{x}) \rightarrow x_1 := e_1(\bar{v}, \bar{x}), \ldots, x_n := e_n(\bar{v}, \bar{x})$ be a guarded command, where $\bar{v}$ and $\bar{x}$ denote the sets of program variables and parameters respectively. We say that the guarded command is *monotone* if $g(\bar{d}_1, \bar{a}_1)$ and $\bar{d}_1 \preceq \bar{d}_2$ imply that there is a $\bar{a}_2$ such that $g(\bar{d}_2, \bar{a}_2)$ and $e_i(\bar{d}_1, \bar{a}_1) \preceq e_i(\bar{d}_2, \bar{a}_2)$, for i: $1 \leqslant i \leqslant n$. A program P is said to be *monotone* if all the guarded commands in P are monotone.

Lemma 8.4. *If a program P is monotone with respect to a preorder $\preceq$, then $[\![P]\!]$ will also be monotone with respect to $\preceq$.*

Proof. The proof follows from the definitions. □

9. Unary constraint systems

In this section, we introduce a class of constraint systems called *unary constraint systems*. Assume a program P with the set $V = \{v_1,\ldots,v_n\}$ of program variables. Assume a set of domains each with a decidable[5] preorder defined on it. Let[6] $\langle d_1,\ldots,d_m \rangle \preceq \langle d'_1, \ldots, d'_m \rangle$ denote $d_i \preceq d'_i$ for i: $1 \leqslant i \leqslant m$. A *unary constraint* is of the form form $\phi_{\bar{d}}$, where $\bar{d} \in dom(v_1) \times \cdots \times dom(v_n)$, and $[\![\phi_{\bar{d}}]\!] = \{\bar{v} : \bar{d} \preceq \bar{v}\}$. Observe $\phi_{\bar{d}_1} \sqsubseteq \phi_{\bar{d}_2}$ if and only if $\bar{d}_1 \preceq \bar{d}_2$, and hence it follows by decidability of $\preceq$ that $\sqsubseteq$ is also decidable.

[3] We assume that the assignment is "type-correct" in the sense that $dom(v_i) = dom(e_i)$, i.e., that the types of v_i and e_i coincide for $i = 1,\ldots,n$.

[4] Observe that each tuple $\langle d_1,\ldots,d_n \rangle \in \langle \mathscr{D}_1 \times \cdots \times \mathscr{D}_n \rangle$ defines a unique state $\sigma \in \Sigma$, where $\sigma(x_i) = d_i$ for $i : 1 \leqslant i \leqslant n$.

[5] By decidability of $\preceq$ we mean that $d_1 \preceq d_2$ can be computed for any d_1 and d_2.

[6] We assume that d_i and d'_i belong to the same domain for $i : 1 \leqslant i \leqslant m$.

Notice that in the case of unary constraint systems the relation $\preceq_{\mathscr{C}}$, derived from the constraint system, coincides with the relation $\preceq$ on the set of states.

A function F is said to be *monotone* if $\bar{y}_1 \preceq \bar{y}_2$ implies $F(\bar{y}_1) \preceq F(\bar{y}_2)$.

In Sections 9.1 and 9.2 we give examples of unary constraint systems. Furthermore, all the programs in these sections are of a certain form (described in the following proposition) which guarantees monotonicity.

Proposition 9.1. *Let P be a program. If each action of P is of form*

$$\bar{d}(\bar{x}) \preceq \bar{f}(\bar{v},\bar{x}) \rightarrow \bar{v} := \bar{e}(\bar{v},\bar{x}),$$

where all functions occurring in the expressions $\bar{f}$ and $\bar{e}$ are monotone, then P is monotone.

Proof. The proof follows from the definitions. □

9.1. Programs over natural numbers

Let $\mathscr{N}$ be the set of natural numbers equipped with the standard less-than or equal ordering $\leqslant$, which is also a well quasi-ordering (Section 7). The relation is extended in the usual way to tuples of natural numbers. We consider programs that operate on a finite set $\{v_1,\ldots,v_n\}$ of variables with domain $\mathscr{N}$. Each state σ is defined by a tuple $\langle d_1,\ldots,d_n\rangle$ of natural numbers. We consider a constraint system $\mathscr{C}_{\mathscr{N}}$, where each constraint in $\mathscr{C}_{\mathscr{N}}$ is of the form $\phi_{\bar{d}}$, denoting $\{\bar{v} : \bar{d} \leqslant \bar{v}\}$. Observe that for finite sets $\Phi_1, \Phi_2 \subseteq \Phi$, we have $\Phi_1 \sqsubseteq \Phi_2$ if and only if for each $\phi_{\bar{d}_2} \in \Phi_2$ there is a constraint $\phi_{\bar{d}_1} \in \Phi_1$ such that $\bar{d}_1 \leqslant \bar{d}_2$. Observe also that the relations $\preceq_{\mathscr{C}_{\mathscr{N}}}$ and $\leqslant$ coincide. It follows that $\mathscr{C}_{\mathscr{N}}$ is well quasi-ordered and is trivially disjunctive.

From Proposition 9.1 we get the monotonicity (with respect to $\preceq_{\mathscr{C}_{\mathscr{N}}}$) for programs, whose actions are of form

$$\bar{d} \leqslant \bar{f}(\bar{x}) \rightarrow v_1,\ldots,v_n := e_1(\bar{x}),\ldots,e_n(\bar{x}),$$

where all occurring functions are monotone. Examples of monotone functions are addition, multiplication, exponentiation, maximum, minimum, and addition/subtraction of a constant.

A particular case of this class of programs are Petri nets, for which it follows that the coverability problem is decidable. But the class is much wider since it allows statements of the form

$$x_1 := x_1 + x_2$$

or even

$$x_1 \geqslant 2 \& x_2 \geqslant 5 \quad \rightarrow \quad x_1 := x_1 * x_2 - 4.$$

9.2. Sequences

Let us now consider programs that employ as domain the set $\mathscr{D}^*$ of sequences of elements in some domain $\mathscr{D}$ with a well quasi-ordering $\preceq$. In Section 7, we showed how a well quasi-ordering $\preceq^*$ on $\mathscr{D}^*$ can be generated in a natural way form $\preceq$.

As a concrete application, let $\mathscr{D}$ be a finite set with $\preceq$ taken to be the identity relation. Assume that a program contains one variable v with $\mathscr{D}^*$ as domain. We consider a constraint system $\mathscr{C}_\mathscr{S}$, where a constraint in $\mathscr{C}_\mathscr{S}$ is of the form ϕ_x with the interpretation $[\![\phi_x]\!] = \{y : x \preceq^* y\}$. Notice that $\mathscr{C}_\mathscr{S}$ is well quasi-ordered and disjunctive.

We can then define actions of the following forms. We let m be a parameter which ranges over $\mathscr{D}$, and x, y be parameters which range over $\mathscr{D}^*$.

- *Send*(m): $v := v \bullet m$
  This action adds an element m at the end of v.
- *Recieve*(m): $x \bullet m \bullet y \preceq v \rightarrow v := y$
  This action is enabled if v contains the element m, in which case it removes all elements in v before and including the element m, and also (nondeterministically) some of the elements after the occurrence of m.
- *Lose*: $x \preceq v \rightarrow v = x$.
  This operation arbitrarily loses some of the elements in v.

The above three types of operations form the basis for the so-called *lossy channel systems* studied in [7]. All three operations can easily be checked to be monotone with respect to $\preceq_{\mathscr{C}_\mathscr{S}}$.

10. Binary constraint systems

In this section, we will investigate another form of constraints, called *binary constraints*, in which two program variables are compared. Recall that a unary constraint essentially relates the value of a program variable to a constant in its domain. Our intention is that a binary constraint should be a predicate which compares the values of two variables in some way.

Assume a program P. Let *dist* be a binary function whose range $\mathscr{D}$ is equipped with a decidable preorder $\preceq$. A *binary constraint* is a formula of form $d \preceq dist(u, u')$, where $d \in \mathscr{D}$ and where u and u' are program variables or parameters. A *polytope* is a conjunction of binary constraints.

Assume a set of binary constraints over the program variables and parameters of P. Let $\mathscr{C}$ be the set of polytopes. According to Definition 6.3, $\mathscr{C}$ gives rise to a preorder $\preceq_\mathscr{C}$ which for each tuple $\bar{u}$ of variables (each of which is either a program variable of parameter), compares tuples $\bar{d}$ of values of $\bar{u}$ according to $\bar{d} \preceq_\mathscr{C} \bar{d}'$ iff $d \preceq dist(d_i, d_j) \implies d \preceq dist(d'_i, d'_j)$ whenever $d \preceq dist(v_i, v_j)$ is a binary constraint in $\mathscr{C}$. As a special case, the restriction of $\preceq_\mathscr{C}$ to values of the tuple $\bar{v}$ of program variables is a constraint system for P.

Let us now give a sufficient criterion for monotonicity of a program with respect to $\preceq_\mathscr{C}$.

Proposition 10.1. *Let P be a program. If each action of P is of form*

$$g(\bar{v}, \bar{x}) \rightarrow \bar{v} := \bar{e}(\bar{v}, \bar{x}),$$

where
- *the guard is a polytope in $\mathscr{C}$, and*
- *each e_i is either a program variable or a parameter, i.e., e_i is either of form v_j or of form x_j for some j, and*
- *for each pair $\bar{d}, \bar{d}'$ of values of the program variables $\bar{v}$ with $\bar{d} \preceq_{\mathscr{C}'} \bar{d}'$, and each possible value $\bar{a}$ of the parameters $\bar{x}$, there is a value $\bar{a}'$ of $\bar{x}$ with $\langle \bar{d}, \bar{a} \rangle \preceq_{\mathscr{C}} \langle \bar{d}', \bar{a}' \rangle$,*

then P is monotone.

Proof. The proof follows from Lemma 8.4. □

We shall now see how Proposition 10.1 can be applied to two specific models of infinite-state systems.

10.1. Relational automata

An example of the use of binary constraints is the class of relational automata, studied by Cerans [17], either with rational numbers or integers as the domain of values. Let us here consider the case of integers.

Let the domain of program variables and parameters be the set $\mathscr{Z}$ of integers. Define a distance function by $dist(v, w) = w - v$. Define $\mathscr{C}$ to be the set of polytopes built from binary constraints of form $d \leqslant dist(u, u')$ where $d \in \mathscr{N}$ is a nonnegative integer and where u, u' are either program variables or parameters. The ordering on states intuitively makes $\bar{d} \preceq_{\mathscr{C}} \bar{d}'$ be true if the values of all variables appear in the same relative order, and if the distance between adjacent variables is at least the same in $\bar{d}'$ as in $\bar{d}$. This ordering is called "sparser than" in [17].

Let integral relational automata be the class of programs that operate on a set of integer-valued variables, and whose actions are of form $g \rightarrow \bar{v} := \bar{e}$ where
- g is a conjunction of statements of form $u + k \leqslant u'$ where u, u' are either program variables, parameters, or integer constants, and where k is a nonnegative integer,
- each e_i in $\bar{e}$ is either a program variable, a parameter, or an integer constant.

The class IRA essentially corresponds to integral relational automata, studied by Cerans in [17]. By Proposition 10.1, each program in IRA is monotone. Furthermore, we note that the constraint system is well quasi-ordered, whence the reachablility problem is decidable.

We note that we can use the same presentation to obtain the class of rational relational automata, simply by changing the domains of variables, but retaining the constraint system. Proposition 10.1 holds also for this model.

10.2. Programs over real-valued clocks

As another application of Proposition 10.1, we will present another class of program that operates on nonnegative real-valued clocks. The class essentially corresponds to the class of timed automata studied in e.g., [2, 1, 15], with some added capability for random assignment. In this presentation, we omit the finite-state control component for simplicity. It can be added without difficulty.

Let the domain of program variables and parameter be the set $\mathscr{R}^{\geqslant}$ of nonnegative reals. Let K be a program-dependent nonnegative integer which denotes the largest constant that syntactically appears in the program. Intuitively, our constraints will compare clocks to each other and to 0, and record the integer part of the difference if the difference is at most K. Differences larger than K need not be distinguished from K. Define a distance function on $\mathscr{R}^{\geqslant}$ by

$$\begin{aligned} dist(v,w) = \text{if } v > K \text{ then } & -\infty \\ & \text{else if } w > K \text{ then } K - v \\ & \qquad \text{else } w - v. \end{aligned}$$

Intuitively, $dist(v,w)$ records the difference $w - v$ if both v and w are between 0 and K. If v or w is larger than K, then we make sure that it does not matter how much larger than K this argument actually is.

Consider a set of variables and parameters. We shall actually consider two constraint systems.

A *binary constraint* is either of form

(1) $d \leqslant dist(v, v')$ where v, v' are program variables and d is an integer with $-K \leqslant d \leqslant K$, or
(2) $u \leqslant d$ where u is a variable (either a program variable or a parameter), and d is an integer with $0 \leqslant d \leqslant K$, or
(3) $d \leqslant u$ where u is a variable (either a program variable or a parameter), and d is an integer with $0 \leqslant d \leqslant K$.

A binary constraint is *downward closed* if it is of one of the first two types. A polytope is *downward closed* if it is the conjunction of downward closed binary constraints. To motivate our distance function, we note that we can write all constraints as binary constraints in the following way: $u \leqslant d$ is equivalent to $-d \leqslant dist(u, 0)$, and $d \leqslant u$ is equivalent to $d \leqslant dist(0, u)$.

Let $\mathscr{C}$ be the set of polytopes, and let $\mathscr{C}'$ be the set of downward closed polytopes. For a tuple $\bar{v} = \langle v_1, \ldots, v_n \rangle$ of program variables and a parameter δ with values in $\mathscr{R}^{\geqslant}$, define $\bar{v} + \delta$ as $\langle v_1 + \delta, \ldots, v_n + \delta \rangle$.

Consider the class CP (for clock programs), whose actions are of form $g(\bar{v} + \delta, \bar{x}) \to \bar{v} := \bar{e}(\bar{v} + \delta, \bar{x})$, where

- $g(\bar{v}, \bar{x})$ is a polytope, and
- each e_i in $\bar{e}(\bar{v} + \delta, \bar{x})$ is either of form $v_j + \delta$ (a program variable with an addition of δ), a parameter, or an integer constant.

We are going to show that the program is monotone with respect to the ordering $\preceq_{\mathscr{C}'}$ on program states. However, we cannot use Proposition 10.1 directly, due to the use of addition in the actions. We therefore separate each action into two parts: first an action which allows "time to pass" by the amount δ, and then an action which performs the assignment of possibly new values to the variables. It turns out that we have to use the stronger preorder $\preceq_{\mathscr{C}}$ to make the second part monotone. The structure of the argument is therefore the following.

(1) We first show that the action $g(\bar{v}+\delta, \bar{x}) \rightarrow \bar{v} := \bar{v}+\delta$ is monotone from $\preceq_{\mathscr{C}'}$ to $\preceq_{\mathscr{C}}$. This means that for any states $\bar{d}_1$, $\bar{d}_2$, and $\bar{d}_3$, with $\bar{d}_1 \preceq_{\mathscr{C}'} \bar{d}_2$ and $\bar{d}_1 \rightarrow \bar{d}_3$, there exists a state $\bar{d}_4$ such that $\bar{d}_3 \preceq_{\mathscr{C}} \bar{d}_4$ and $\bar{d}_2 \rightarrow \bar{d}_4$, where we consider transitions derived from this action.

(2) By Proposition 10.1, we see that the action $g(\bar{v}, \bar{x}) \rightarrow \bar{v} := \bar{e}(\bar{v}, \bar{x})$ is monotone with respect to $\preceq_{\mathscr{C}}$.

It follows that the action $g(\bar{v}+\delta, \bar{x}) \rightarrow \bar{v} := \bar{e}(\bar{v}+\delta, \bar{x})$ is monotone from $\preceq_{\mathscr{C}'}$ to $\preceq_{\mathscr{C}}$. Since the preorder $\preceq_{\mathscr{C}}$ is stronger than $\preceq_{\mathscr{C}'}$, it follows that $g(\bar{v}+\delta, \bar{x}) \rightarrow \bar{v} := \bar{e}(\bar{v}+\delta, \bar{x})$ is monotone with respect to $\preceq_{\mathscr{C}'}$. Since $\mathscr{C}'$ is well quasi-ordered, we can use this constraint system to verify reachability for clock programs.

11. Conclusion

In this paper, we have considered symbolic reachability analysis for infinite-state systems, and presented conditions under which this analysis is guaranteed to terminate. From these general conditions, we have derived several computation models in which the reachability problem is decidable. These models have state variables that range over many different domains, such as natural numbers, multisets, unbounded finite sequences, integers, and real numbers. Systems for which reachability can be analyzed using our methods include Petri Nets [26, 27], Lossy Channel Systems [6], relational automata [17], timed automata [2], and unbounded networks of process with clock variables [8]. The general conditions have been presented in our earlier work [5], and in this paper we have shown in more detail how to find new computation models that satisfy the conditions. We hope to have conveyed the conclusion that the conditions are very natural, and cover many situations which are still to be investigated.

Acknowledgements

This work owes much to our Karlis Čerāns and Yih-Kuen Tsay, who collaborated in developing the general framework of [5]. Thanks also to Tatiana Rybina for contributions to the modeling and analysis of the Future-bus protocol.

References

[1] R. Alur, T. Henzinger, A really temporal logic, Proc. 30th Ann. Symp. Foundations of Computer Science, 1989, pp. 164–169.

[2] R. Alur, C. Courcoubetis, D. Dill, Model-checking for real-time systems, Proc. 5th IEEE Internat. Symp. on Logic in Computer Science, Philadelphia, 1990, pp. 414–425.

[3] P.A. Abdulla, B. Jonsson, Verifying programs with unreliable channels, Proc. 8th IEEE Internat. Symp. on Logic in Computer Science, 1993, pp. 160–170.
[4] P.A. Abdulla, M. Kindahl, Decidability of simulation and bisimulation between lossy channel systems and finite state systems, in: Lee, Smolka (Eds.), Proc. CONCUR '95, 6th Internat. Conf. on Concurrency Theory, Lecture Notes in Computer Science, vol. 962, Springer, Berlin, 1995, pp. 333–347.
[5] P.A. Abdulla, K. Čerāns, B. Jonsson, T. Yih-Kuen, General decidability theorems for infinite-state systems, Proc. 11th IEEE Internat. Symp. on Logic in Computer Science, 1996, pp. 313–321.
[6] P.A. Abdulla, B. Jonsson, Undecidable verification problems for programs with unreliable channels, Inform. and Comput. 130 (1) (1996) 71–90.
[7] P.A. Abdulla, B. Jonsson, Verifying programs with unreliable channels, Inform. and Comput. 127 (2) (1996) 91–101.
[8] P.A. Abdulla, B. Jonsson, Verifying networks of timed processes, in: B. Steffen (Ed.), Proc. TACAS '98, 7th Internat. Conf. on Tools and Algorithms for the construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 1384, Springer, Berlin, 1998, pp. 298–312.
[9] R.J.R. Back, R. Kurki-Suonio, Distributed cooperation with action systems, ACM Trans. Programming Languages Systems 10 (4) (1988) 513–554.
[10] J.M. Barzdin, J.J. Bicevskis, A.A. Kalninsh, Automatic construction of complete sample systems for program testing, IFIP Congress, 1977.
[11] J. Bradfield, C. Stirling, Local model checking for infinite state spaces, Theoret. Comput. Sci. 96 (1992) 157–174.
[12] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comput. C-35 (8) (1986) 677–691.
[13] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Proc. 5th IEEE Internat. Symp. on Logic in Computer Science, 1990.
[14] O. Burkart, B. Steffen, Composition, decomposition, and model checking of pushdown processes, Nordic J. Comput. 2 (2) (1995) 89–125.
[15] K. Čerāns, Decidability of bisimulation equivalence for parallel timer processes, Proc. Workshop on Computer Aided Verification, Lecture Notes in Computer Science, vol. 663, Springer, Berlin, 1992, pp. 302–315.
[16] K. Čerāns, Feasibility of finite and infinite paths in data dependent programs, LFCS'92, Lecture Notes in Computer Science, vol. 620, Springer, Berlin, 1992, pp. 69–80.
[17] K. Čerāns, Deciding properties of integral relational automata, in: Abiteboul, Shamir (Eds.), Proc. ICALP'94, Lecture Notes in Computer Science, vol. 820, Springer, Berlin, 1994, pp. 35–46.
[18] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988.
[19] L.E. Dickson, Finiteness of the odd perfect and primitive abundant numbers with *n* distinct prime factors, Amer. J. Math. 35 (1913) 413–422.
[20] A. Finkel, Reduction and covering of infinite reachability trees, Inform. and Comput. 89 (1990) 144–179.
[21] A. Finkel, Decidability of the termination problem for completely specified protocols, Distrib. Comput. 7 (3) (1994).
[22] S.M. German, A.P. Sistla, Reasoning about systems with many processes, J. ACM 39 (3) (1992) 675–735.
[23] P. Godefroid, P. Wolper, Using partial orders for the efficient verification of deadlock freedom and safety properties, Formal Methods System Des. 2 (2) (1993) 149–164.
[24] T.A. Henzinger, Hybrid automata with finite bisimulations, Proc. ICALP '95, 1995.
[25] G. Higman, Ordering by divisibility in abstract algebras, Proc. London Math. Soc. 2 (1952) 326–336.
[26] P. Jančar, Decidability of a temporal logic problem for Petri nets, Theoret. Comput. Sci. 74 (1990) 71–93.
[27] P. Jančar, F. Moller, Checking regular properties of Petri nets, Proc. CONCUR '95, 6th Internat. Conf. on Concurrency Theory, 1995, pp. 348–362.
[28] B. Jonsson, J. Parrow, Deciding bisimulation equivalences for a class of non-finite-state programs, Inform. and Comput. 107 (2) (1993) 272–302.
[29] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shahar, Symbolic model checking with rich assertional languages, in: O. Grumberg (Ed.), Proc. 9th Internat. Conf. on Computer Aided Verification, Haifa, Israel, vol. 1254, Springer, Berlin, 1997, pp. 424–435.

[30] D. Peled, Combining partial order reductions with on-the-fly model checking, Formal Aspects Comput. 8 (1996) 39–64.

[31] J.M. Rushby, Specification, proof checking, and model checking for protocols and distributed systems with PVS, Tutorial notes for FORTE/PSTV ’97, Nov. 1997.

[32] N. Shankar, PVS: Combining specification, proof checking, and model checking, Proc. FMCAD ’96, Lecture Notes in Computer Science, vol. 1166, Springer, Berlin, 1996.

[33] C. Stirling, Decidability of bisimulation equivalence for normed pushdown processes, Proc. CONCUR ’96, 7th Internat. Conf. on Concurrency Theory, Lecture Notes in Computer Science, vol. 1119, Springer, Berlin, 1996, pp. 217–232.

[34] A. Valmari, On-the-fly verification with stubborn sets in: Courcoubetis (Ed.), Proc. 5th Internat. Conf. on Computer Aided Verification, Lecture Notes in Computer Science, vol. 697, Springer, Berlin, 1993, pp. 59–70.

[35] M.Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, Proc. 1st IEEE Internat. Symp. on Logic in Computer Science, June 1986, pp. 332–344.

[36] P. Wolper, Expressing interesting properties of programs in propositional temporal logic (extended abstract), Proc. 13th ACM Symp. on Principles of Programming Languages, Jan. 1986, pp. 184–193.