The Algorithmic Verification of Reactive Systems

Lecture 10:

Algorithms for Systems with Infinite State Spaces

The Problem

- If a FCS has variables with unbounded domains, there is no guarantee that its state space is finite.
- When the state space is indeed infinite, it cannot be computed by the type of enumerative search we have considered so far.
- It might, however, be possible to compute a *symbolic* (formula based) representation of the state space.
- Note that if the symbolic representation is itself in a decidable language, which is essential for it to be useful, it cannot always be computed since nontrivial state-space properties are in general undecidable.

Two Strategies

- A first strategy is to identify restricted classes of systems with infinite state spaces for which the state space can always be computed.
- A second strategy is to consider undecidable classes of systems, but to be satisfied with partial algorithms, i.e. algorithms that may fail to give an answer or run forever.
- We will examine examples of both strategies.

The Starting Point

- We will consider systems that are modelled by FCSs with a single process.
- This is not restrictive (one can always compute the concurrent execution of the finite control of a set of processes).
- The domain of the memory variables and the operations that can be performed upon them are crucial here, so we will define them carefully.

We thus adopt a simplified notation and define a system as a tuple $(C, c_0, M, m_0, \mathit{Op}, \Delta)$, where

- C is a finite set of *control locations*,
- M is a (possibly infinite) *memory domain* (often given as the cross product of the domains of a finite number of variables),
- $\mathit{Op} \subseteq 2^{M \to M}$ is a set of *memory operations*,
- $\Delta \subseteq C \times \mathit{Op} \times C$ is a finite set of *transitions*,
- c_0 is an *initial control location*, and m_0 is an *initial memory content*.

A state is thus an element of $C \times M$

Pushdown Systems

In a pushdown system, the memory is a single stack. Formally,

- The memory domain M is Σ^* for a finite alphabet Σ.
- The memory operations are a_+ and a_- for every $a \in \Sigma$. They are defined by $a_+(w) = wa$ and $a_-(wa) = w$ for every $w \in \Sigma^*$ (the value of $a_-(w)$ is not defined if w does not end with the symbol a).

The reachable states of a pushdown system can be computed as follows.

Given a pushdown system $A = (C, c_0, M, m_0, \mathit{Op}, \Delta)$ with $M = \Sigma^*$, one constructs the *reachability automaton* of A.

It is the finite-state automaton $A_r = (Q_r, \Sigma_r, \Delta_r, q_r^0, F_r)$ such that

- The set of states Q_r is identical to C;
- The input alphabet Σ_r is identical to Σ;
- The transition relation $\Delta_r \subseteq Q_r \times (\Sigma_r \cup \{\varepsilon\}) \times Q_r$ is the smallest relation that satisfies the following conditions, where Δ_r^* denotes the reflexive and transitive closure of Δ_r, and ε denotes the empty word:
  - If $(q, a_+, q') \in \Delta$, then $(q, a, q') \in \Delta_r$, and
  - If $(q, a_-, q') \in \Delta$ and $(q'', a, q) \in \Delta_r^*$, then $(q'', \varepsilon, q') \in \Delta_r$;
- The initial state q_r^0 is c_0;
- All the states are accepting, i.e., we have $F_r = Q_r$.

A_r characterizes the reachable states. A system state (q, w) is reachable iff q is reachable in A_r through the word w.

Systems with Unbounded FIFO Queues

- Finite-state systems communicating through unbounded queues are a very common model for concurrent systems.
- Since we are not focussing on the concurrency aspects of this model, we will consider a single finite-state system using a number of unbounded queues as memory
- Note that it is possible to simulate the tape of a Turing machine with just a single queue. Thus problems about the state space of systems with unbounded FIFO queues are in general undecidable.

Unbounded FIFO Queues: The Model

In a *queue system*, the memory domain is a set of unbounded queues. Formally, we have the following.

- The memory domain is of the form $\Sigma_1^* \times \Sigma_2^* \times \cdots \times \Sigma_n^*$, where $n > 0$ represents the *number of queues*, and each Σ_i is the finite *queue alphabet* of the i-th queue q_i (we assume they are distinct).

- The set of memory operations *Op* contains the two *queue operations* $q_i!a$ and $q_i?a$ for each queue q_i and symbol $a \in \Sigma_i$.

  The *send* operation $q_i!a$ is defined by $(q_i!a)(w_1, \ldots, w_{i-1}, w_i, w_{i+1}, \ldots, w_n) = (w_1, \ldots, w_{i-1}, w_i a, w_{i+1}, \ldots, w_n)$.

  The *receive* operation $q_i?a$ is defined by $(q_i?a)(w_1, \ldots, w_{i-1}, aw_i, w_{i+1}, \ldots, w_n) = w_1, \ldots, w_{i-1}, w_i, w_{i+1}, \ldots, w_n)$ (it is not defined if the content of q_i does not start with the symbol a).

Analyzing Queue Systems

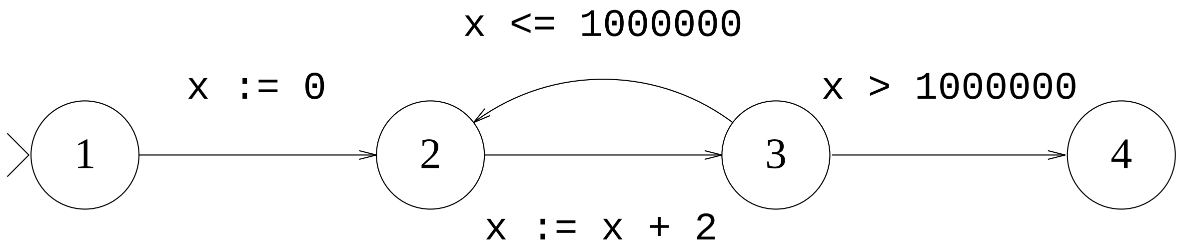
To compute the state space of queue systems we need two things.

- The first is a finite representation of possibly infinite sets of queue contents.
- The second is a technique for finitely computing the representation of an infinite number of reachable states. Indeed, if one just does a traditional state space search, one only generates a single state at a time and thus one is unable to ever generate an infinite set of states.

Generating Infinite Sets of States: The Loop First Search

- Identify some loops in the finite-state control of the system.
- Explore the state space as usual, but when reaching a loop *first* attempt to compute the effect of indefinitely iterating the sequence of operations labeling the loop.
- When this computation succeeds, carry on the state space exploration, handling simultaneously all data values that are possible for each control point.
- The state-space exploration terminates when nothing can be added to the computed state space.

An example of a loop-first search



$(1, \perp)$
$(2, 0)$
$(2, 2k)$ with $0 \leq k \in \mathbf{N} \leq 500000$
$(3, 2k+2)$ with $0 \leq k \in \mathbf{N} \leq 500000$
$(4, 1000002)$

Note that a loop first search allows one to go arbitrarily deep into a computation in one step.

The loop first search can fail for at least two reasons:

- It does not terminate,
- Some cycles cannot be handled.

Representing the Content of Queues: The QDD

A *Queue Decision Diagram (QDD)* is a finite automaton representation of a set of queue contents.

- A content $(w_1, \ldots, w_n)$ for a queue system with n queues is represented by the concatenation $w_1 \cdot w_2 \cdots w_n$ of the individual queue contents taken in a fixed order.

- A QDD is a finite automaton over the union of the queue $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n$ of the queue alphabets such that all words accepted by the automaton satisfy

$$w = w|_{\Sigma_1} w|_{\Sigma_2} \ldots w|_{\Sigma_n}.$$

  That is, every word accepted by the automaton can be interpreted as a content for the set of queues of the system.

Theorem
The queue contents representable by QDDs are exactly the recognizable languages, i.e. are subsets of $\prod_{1 \leq i \leq n} \Sigma_i^*$ of the form

$$\bigcup_{1 \leq j \leq k} \prod_{1 \leq i \leq n} L_{ij},$$

where each L_{ij} is a regular subset of M_i^*.

Operations on QDDs

A state of a queue system is a pair $(c, m) \in C \times M$. We consider sets of states with an identical control location c represented as a pair (c, A) where A is a QDD. We have to address the following problems.

- Compute the effect of applying a transition (c, op, c') to the states represented by (c, A), i.e. the set of states $\{(c', m') | \exists m(m \in L(A) \wedge m' \in op(m)\}$.
- Compute the effect of applying a sequence of transitions to (c, A).
- Compute the effect of repeatedly applying a cyclic sequence of transitions to (c, A).

The hard part of these problems is, of course, to compute (if it exists) the QDD resulting from the application of the operations.

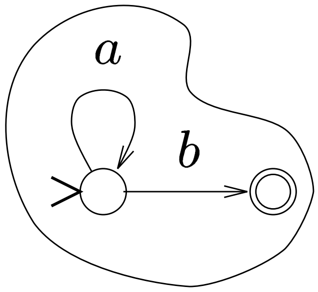
Applying Operations to QDDs
The Single Queue Case

Consider a system with a single queue. We are given a QDD $A = (S, \Sigma, \delta, S_0, F)$ and want to compute a QDD A' such that $L(A') = \{op(m) | m \in L(A)\}$ where *Op* is $q?a$ or $q!a$ for $a \in \Sigma$.

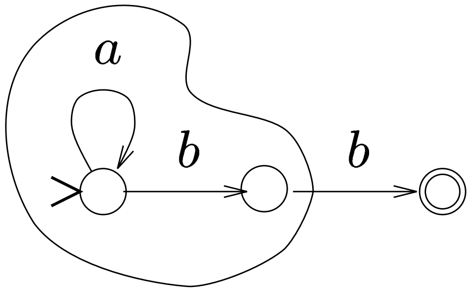
- For the operation $q?a$, we have $A' = (S, \Sigma, \delta, S'_0, F)$ with $S'_0 = \{s' | s' \in \delta(s, a)$ for $s \in S_0\}$
- For the operation $q!a$, we have $A' = (S', \Sigma, \delta', S_0, F')$ with
  - $S' = S \cup \{f\}$, $F' = \{f\}$,
  - $\delta'(s, \alpha) = \delta(s, \alpha)$ for $s \notin F$ or $\alpha \neq a$, and $\delta'(s, \alpha) = \delta(s, \alpha) \cup \{f\}$ for $s \in F$ and $a = \alpha$,
- For a sequence of operations $\sigma = op_1 op_2 \ldots op_k$ the resulting QDD is obtained by successively applying the single operation constructions for op_1, op_2 ....

An Example

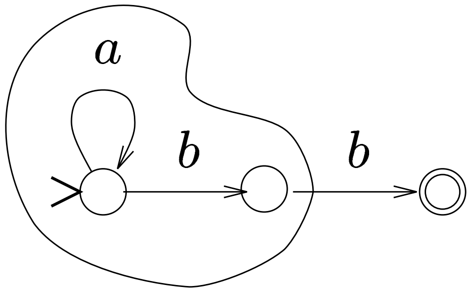
a^*b:



$(q!b)[a^*b]$:



$(q!b; q?a)[a^*b]$:



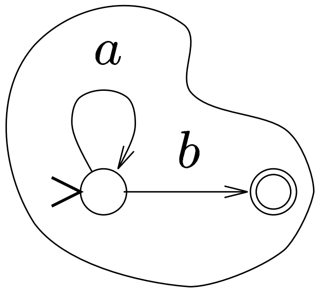
Iterating Sequences of Operations on QDDS

To compute the effect of iterating a sequence of operation σ to the set of queue contents represented by a QDD A, i.e. to compute $\sigma^*(A)$, we proceed as follows

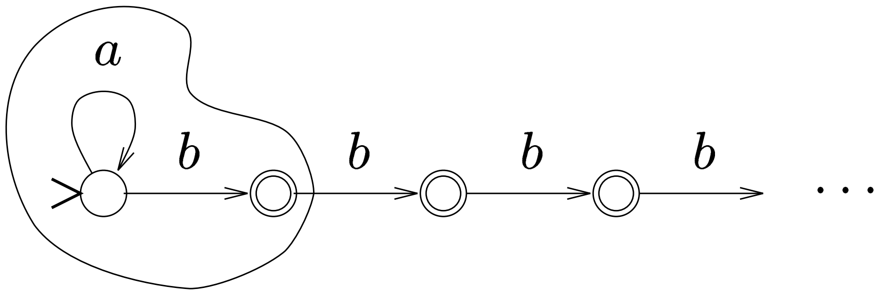
- We use $\sigma^*(A) = \bigcup_k \sigma^k(A)$
- Some periodicity will eventually occur within the $\sigma^k(A)$.
- $\sigma^*(A)$ can thus be represented by a finite union.

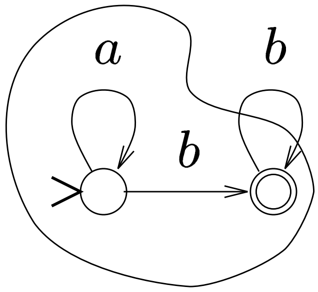
Iterating Sequences on a QDD: An Example

a^*b:



$(q!b; q?a)^*[a^*b]$:





Operations on Systems with Multiple Queues

- For single operations, one simply operates as above on the part of the QDD representing the queue on which the operation is performed.
- Sequences of operations can be similarly handled.
- The result of iterating a sequence of operations cannot always be represented as a QDD. For instance $(q_1!a; q_2!b)^*$.

  The problem comes from the ability to count the number of iterations by looking at the content of two or more queues.
- When only one queue allows to count the number of iterations, one can combine the result of handling separately the different queues.

Lossy Queue Systems

- Lossy queue systems are queue systems in which, at any point, any part of the content of a queue can disappear (be lost).
- Very interestingly, questions that are undecidable for queue systems become decidable for lossy queue systems.
- An example of such a property is *state reachability* (is a given state reachable).

State Reachability for Lossy Queue Systems

- The algorithm works by performing a backwards search from the state for which reachability should be decidable.
- The search does not use cycle meta-transitions, but rather “completes” the queue contents at each step.
- “Completing” queue contents means that if

  $$w = a_1 a_2 \ldots a_n$$

  is a queue content, then so are all words in

  $$\Sigma^* a_1 \Sigma^* a_2 \Sigma^* \ldots \Sigma^* a_n \Sigma^*$$

- Note that QDDs can be used for a search in which queue contents are “completed” at each step. Indeed, the required operation can be implemented on the QDD representation of queue contents.

Lossy Queue Systems: Termination

The interesting fact is that the backwards search we have described always terminates. Let us prove it.

- Define $v \preceq w$ if v is a subword of w.
- For the backwards search not to terminate, there must be a control location c that is reached infinitely often with a sequence of queue contents

  $$w_1, w_2, \ldots w_i, \ldots, w_j, \ldots$$

  such that for all $i \leq j$, $w_i \not\preceq w_j$.

  Indeed, if $w_i \preceq w_j$, then w_j is in the "completion" of w_i and the search would terminate when reaching w_j.
- By Higman's theorem, an infinite sequence

  $$w_1, w_2, \ldots w_i, \ldots, w_j, \ldots$$

  such that for all $i \leq j$, $w_i \not\preceq w_j$ cannot exist.

Linear Integer Systems

In a *Linear Integer System*, the memory is a set of unbounded integer variables. Formally, we have the following.

- The memory domain M is $\mathbf{Z}^n$, where $n > 0$ represents the *number of variables*.
- The set of memory operations *Op* contains all functions $M \to M$ of the form

  $$P\vec{x} \leq \vec{q} \to \vec{x} := T\vec{x} + \vec{b}$$

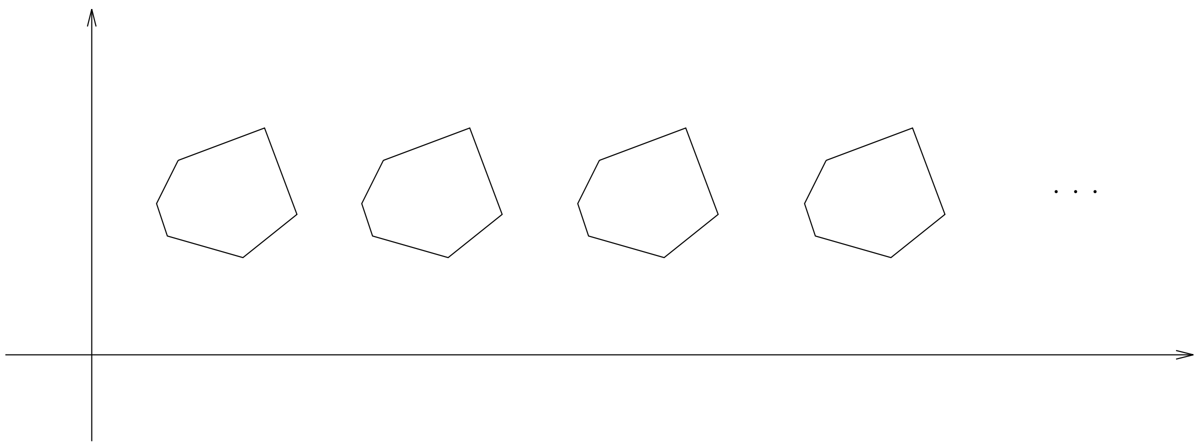
  where $P \in \mathbf{Z}^{m\times n}, \vec{q} \in \mathbf{Z}^m, m \in \mathbf{N}, T \in \mathbf{Z}^{n\times n}$ and $\vec{b} \in \mathbf{Z}^n$.

  The system $P\vec{x} \leq \vec{q}$ is the *guard* of the operation and the transformation $\vec{x} := T\vec{x} + \vec{b}$ is the *assignment* of the operation.

Representing sets of integer values

First idea : linear constrained sets.

- Yes, but iterating a simple operation like $x := x + 3$ yields sets which are periodic unions of linear constrained sets



- One needs means to represent periodicity !

Representing sets of integers II

- Use a logical formalism, e.g. Presburger Arithmetic (first-order arithmetic without multiplication).

  $$\exists k\, x_0(x = x_0 + 5k \wedge 1 \leq x_0 \leq 3 \\ \wedge 2 \leq y \leq 4)$$

  - Expressiveness is sufficient,
  - The problem is computing with such a logical representation.

- Alternative : use automata to represent sets of integers.

Representing set of integers
Encoding numbers

- Binary representation,
- Unbounded numbers,
- Most significant bit first.
- 2's complement for negative numbers (at least p pits for a number x such that $-2^{p-1} \leq x < 2^{p-1}$).

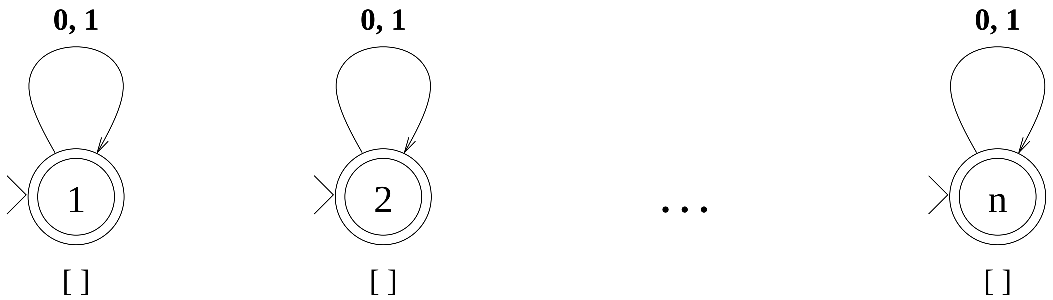
Examples :

$$\begin{array}{rcl} 4 & : & 0100, 00100, 000100, \ldots \\ -4 & : & 100, 1100, 11100, \ldots \end{array}$$

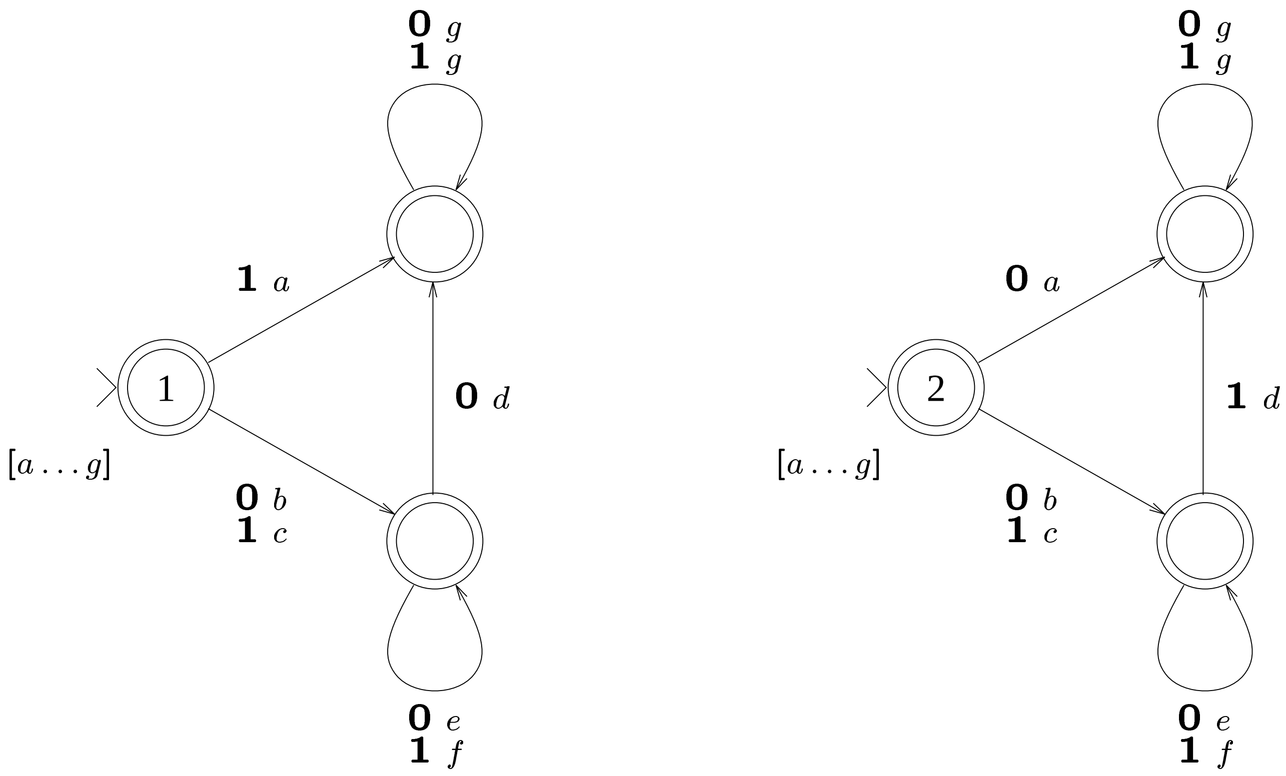
The encodings of any Presburger definable set of integer vectors are accepted by finite automata and computing with sets of integers is reduced to computing with finite automata.

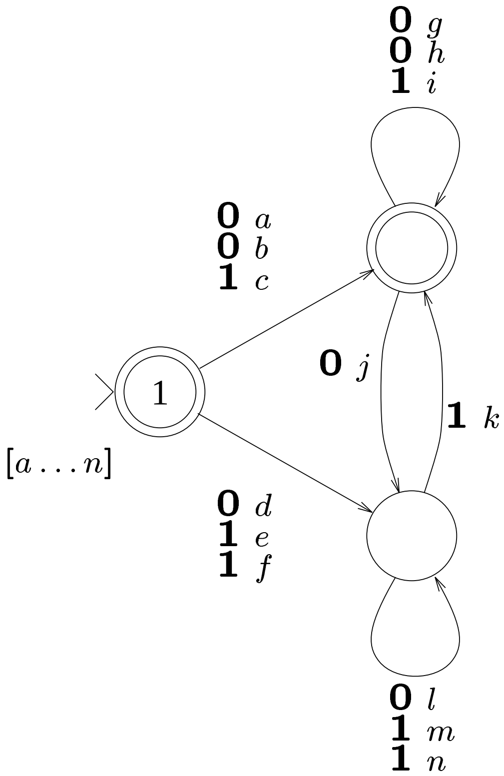
Automata for Elementary predicates

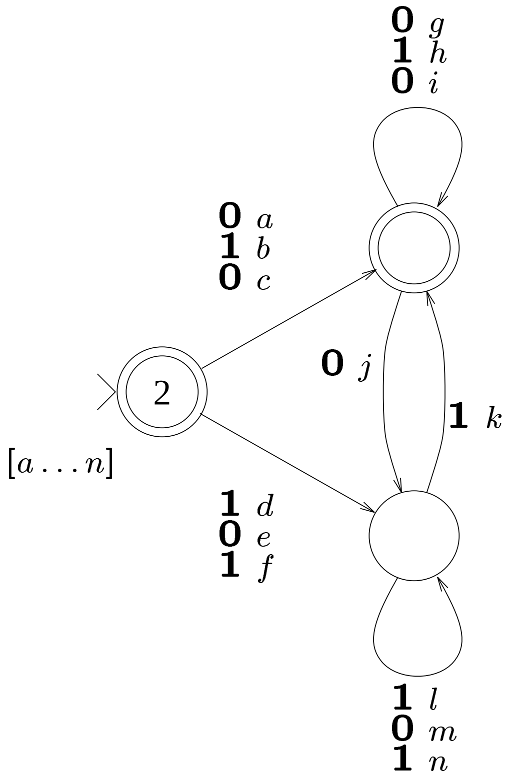
$A_{\mathbf{Z}^n}$

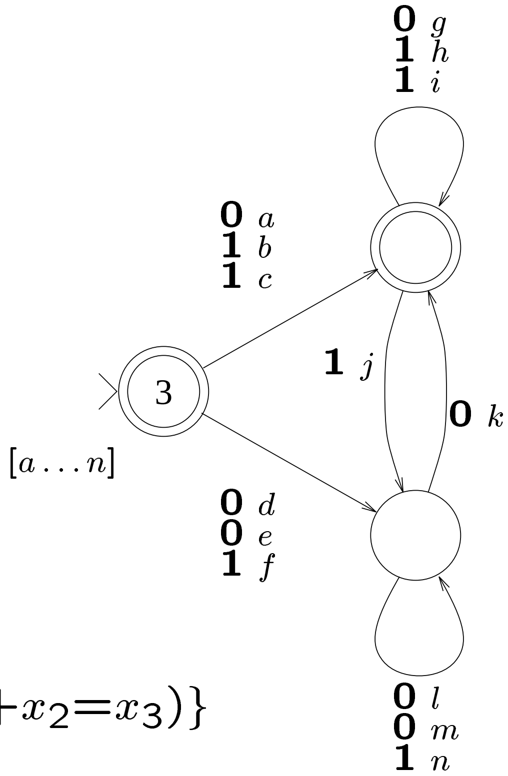


$A_{\{(x_1,x_2)|x_1 \leq x_2\}}$









$A_{\{(x_1,x_2,x_3)|x_1+x_2=x_3)\}}$

Iterating operations on integers

A simple case.

- An instruction $I \equiv T\vec{x} \leq \vec{u} \rightarrow \vec{x} := A\vec{x} + \vec{b}$, with A idempotent ($A^2 = A$)
- An initial value $\vec{x}_0$

Compute the values obtained by the repeated execution of I on $\vec{x}_0$:

$$\begin{array}{ccrcc}
A\vec{x}_0 & + & & + & \vec{b} \\
A\vec{x}_0 & + & A\vec{b} & + & \vec{b} \\
A\vec{x}_0 & + & 2A\vec{b} & + & \vec{b} \\
 & \vdots & & \vdots & \\
A\vec{x}_0 & + & kA\vec{b} & + & \vec{b} \\
 & \vdots & & \vdots &
\end{array}$$

Cycle precondition : $T(A\vec{x}_0 + kA\vec{b} + \vec{b}) \leq \vec{u}$

More general results by B. Boigelot.

Conclusions

- Computing a symbolic representation of the state space of a system is in fact computing the strongest *invariant* for this system.
- However, the focus is now on the computational aspects of obtaining such an invariant.
- Just as in the finite-state case, representation formalisms with which one can compute efficiently will be the key to successes.