Automata-Theoretic Approach to Model Checking

(Based on [Clarke et al. 1999] and [Holzmann 2003])

Yih-Kuen Tsay

Dept. of Information Management
National Taiwan University

Outline

Büchi Automata

- The simplest computation model for finite behaviors is the finite state automaton, which accepts finite words.
- The simplest computation model for infinite behaviors is the ω-automaton, which accepts infinite words.
- Both have the same syntactic structure.
- Model checking traditionally deals with non-terminating systems.
- Infinite words conveniently represent the infinite behaviors exhibited by a non-terminating system.
- Büchi automata are the simplest kind of ω-automata.
- They were first proposed and studied by J.R. Büchi in the early 1960's, to devise decision procedures for the logic S1S.

Büchi Automata (cont.)

- A Büchi automaton (BA) has the same structure as a finite state automaton (FA) and is also given by a 5-tuple $(\Sigma, Q, \Delta, q_0, F)$:
  1. Σ is a finite set of symbols (the *alphabet*),
  2. Q is a finite set of *states*,
  3. $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*,
  4. $q_0 \in Q$ is the *start* (or *initial*) state (sometimes we allow multiple start states, indicated by Q_0 or Q^0), and
  5. $F \subseteq Q$ is the set of *accepting* (final in FA) states.
- Let $B = (\Sigma, Q, \Delta, q_0, F)$ be a BA and $w = w_1 w_2 \ldots w_i w_{i+1} \ldots$ be an infinite string (or word) over Σ.
- A *run* of B over w is a sequence of states $r_0, r_1, r_2, \ldots, r_i, r_{i+1}, \ldots$ such that
  1. $r_0 = q_0$ and
  2. $(r_i, w_{i+1}, r_{i+1}) \in \Delta$ for $i \geq 0$.

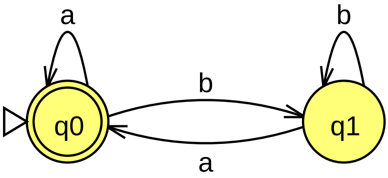
Büchi Automata (cont.)

- Let $\mathit{inf}(\rho)$ denote the set of states occurring infinitely many times in a run ρ.
- A run ρ is *accepting* if it satisfies the following condition:

$$\mathit{inf}(\rho) \cap F \neq \emptyset.$$

- An infinite word $w \in \Sigma^\omega$ is *accepted* by a BA B if there exists an accepting run of B over w.
- The *language* recognized by B (or the language of B), denoted $L(B)$, is the set of all words accepted by B.

An Example Büchi Automaton



- This Büchi automaton accepts infinite words over $\{a, b\}$ that have infinitely many a's.
- Using an ω-regular expression, its language is expressed as $(b^* a)^\omega$.

Closure Properties

- A class of languages is closed under intersection if the intersection of any two languages in the class remains in the class.
- Analogously, for closure under complementation.

Theorem

The class of languages recognizable by Büchi automata is closed under **intersection** *and* **complementation** *(and hence all boolean operations).*

- Note: the theorem would not hold if we were restricted to *deterministic* Büchi automata, unlike in the classic case.

Generalized Büchi Automata

- A generalized Büchi automaton (GBA) has an acceptance component of the form $F = \{F_1, F_2, \cdots, F_n\} \subseteq 2^Q$.
- A run ρ of a GBA is accepting if for each $F_i \in F$, $\mathit{inf}(\rho) \cap F_i \neq \emptyset$.
- GBA's naturally arise in the modeling of finite-state concurrent systems with fairness constraints.
- They are also a convenient intermediate representation in the translation from a linear temporal formula to an equivalent BA.
- There is a simple translation from a GBA to a Büchi automaton, as shown next.

GBA to BA

- Let $B = (\Sigma, Q, \Delta, q_0, F)$, where $F = \{F_1, \cdots, F_n\}$, be a GBA.
- Construct $B' = (\Sigma, Q \times \{0, \cdots, n\}, \Delta', \langle q_0, 0\rangle, Q \times \{n\})$.
- The transition relation Δ' is constructed such that $(\langle q, x\rangle, a, \langle q', y\rangle) \in \Delta'$ when $(q, a, q') \in \Delta$ and x and y are defined according to the following rules:
  - If $q' \in F_i$ and $x = i - 1$, then $y = i$.
  - If $x = n$, then $y = 0$.
  - Otherwise, $y = x$.
- Claim: $L(B') = L(B)$.

Theorem

For every GBA B, there is an equivalent BA B' such that $L(B') = L(B)$.

Model Checking Using Automata

- Kripke structures are the most commonly used model for concurrent and reactive systems in model checking.
- Let AP be a set of atomic propositions.
- A Kripke structure M over AP is a four-tuple $M = (S, R, S_0, L)$:
  1. S is a finite set of states.
  2. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
  3. $S_0 \subseteq S$ is the set of initial states.
  4. $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

Model Checking Using Automata (cont.)

- Finite automata can be used to model concurrent and reactive systems as well.
- One of the main advantages of using automata for model checking is that both the modeled system and the specification are represented in the same way.
- A Kripke structure directly corresponds to a Büchi automaton, where all the states are accepting.
- A Kripke structure (S, R, S_0, L) can be transformed into an automaton $A = (\Sigma, S \cup \{\iota\}, \Delta, \iota, S \cup \{\iota\})$ with $\Sigma = 2^{AP}$ where
  - $(s, \alpha, s') \in \Delta$ for $s, s' \in S$ iff $(s, s') \in R$ and $\alpha = L(s')$ and
  - $(\iota, \alpha, s) \in \Delta$ iff $s \in S_0$ and $\alpha = L(s)$.

Model Checking Using Automata (cont.)

- The given system is modeled as a Büchi automaton A.
- Suppose the desired property is originally given by a linear temporal formula f.
- Let B_f (resp. $B_{\neg f}$) denote a Büchi automaton equivalent to f (resp. $\neg f$); we will later study how a temporal formula can be translated into an automaton.
- The model checking problem $A \models f$ is equivalent to asking whether

$$L(A) \subseteq L(B_f) \text{ or } L(A) \cap L(B_{\neg f}) = \emptyset.$$

- The well-used model checker SPIN, for example, adopts this automata-theoretic approach.
- So, we are left with two basic problems:
  - Compute the intersection of two Büchi automata.
  - Test the emptiness of the resulting automaton.

Intersection of Büchi Automata

- Let $B_1 = (\Sigma, Q_1, \Delta_1, Q_1^0, F_1)$ and $B_2 = (\Sigma, Q_2, \Delta_2, Q_2^0, F_2)$.
- We can build an automaton for $L(B_1) \cap L(B_2)$ as follows.
- $B_1 \cap B_2 = (\Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta, Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times Q_2 \times \{2\})$.
- We have $(\langle r, q, x\rangle, a, \langle r', q', y\rangle) \in \Delta$ iff the following conditions hold:
    - $(r, a, r') \in \Delta_1$ and $(q, a, q') \in \Delta_2$.
    - The third component is affected by the accepting conditions of B_1 and B_2.
        - If $x = 0$ and $r' \in F_1$, then $y = 1$.
        - If $x = 1$ and $q' \in F_2$, then $y = 2$.
        - If $x = 2$, then $y = 0$.
        - Otherwise, $y = x$.
- The third component is responsible for guaranteeing that accepting states from both B_1 and B_2 appear infinitely often.

Intersection of Büchi Automata (cont.)

- A simpler intersection may be obtained when all of the states of one of the automata are accepting.
- Assuming all states of B_1 are accepting and that the acceptance set of B_2 is F_2, their intersection can be defined as follows:

$$B_1 \cap B_2 = (\Sigma, Q_1 \times Q_2, \Delta', Q_1^0 \times Q_2^0, Q_1 \times F_2)$$

where $(\langle r, q \rangle, a, \langle r', q' \rangle) \in \Delta'$ iff $(r, a, r') \in \Delta_1$ and $(q, a, q') \in \Delta_2$.

Checking Emptiness

- Let ρ be an accepting run of a Büchi automaton $B = (\Sigma, Q, \Delta, Q^0, F)$.
- Then, ρ contains infinitely many accepting states from F.
- Since Q is finite, there is some suffix ρ' of ρ such that every state on it appears infinitely many times.
- Each state on ρ' is reachable from any other state on ρ'.
- Hence, the states in ρ' are included in a strongly connected component.
- This component is reachable from an initial state and contains an accepting state.

Checking Emptiness (cont.)

- Conversely, any strongly connected component that is reachable from an initial state and contains an accepting state generates an accepting run of the automaton.
- Thus, checking nonemptiness of $L(B)$ is equivalent to finding a strongly connected component that is reachable from an initial state and contains an accepting state.
- That is, the language $L(B)$ is nonempty iff there is a reachable accepting state with a cycle back to itself.

Double DFS Algorithm

procedure *emptiness*
  for all $q_0 \in Q^0$ **do**
    dfs1(q_0);
  terminate(*True*);
end procedure

procedure *dfs1*(q)
  local q';
  hash(q);
  for all successors q' of q **do**
    if q' not in the hash table **then** *dfs1*(q');
  if *accept*(q) **then** *dfs2*(q);
end procedure

Double DFS Algorithm (cont.)

procedure *dfs2*(q)
  local q';
  flag(q);
  for all successors q' of q **do**
    if q' on *dfs1* stack **then** terminate(*False*);
    else if q' not flagged **then** *dfs2*(q');
    end if;
end procedure

Correctness

Lemma

Let q be a node that does not appear on any cycle. Then the DFS algorithm will backtrack from q only after all the nodes that are reachable from q have been explored and backtracked from.

This lemma still holds for the first DFS in the double DFS algorithm.

Theorem

The double DFS algorithm returns a counterexample for the emptiness of the checked automaton B exactly when the language $L(B)$ is not empty.

Correctness (cont.)

- Suppose a second DFS is started from a state q and there is a path from q to some state p on the search stack of the first DFS.
- There are two cases:
  - There exists a path from q to a state on the search stack of the first DFS that contains only *unflagged* nodes when the second DFS is started from q.
  - On every path from q to a state on the search stack of the first DFS, there exists a state r that is already flagged.
- The algorithm will find a cycle in the first case.
- We show next that the second case is impossible.

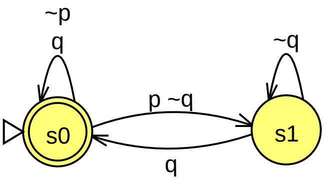
Correctness (cont.)

- Suppose the contrary: on every path from q to a state on the search stack of the first DFS, there exists a state r that is already flagged.
- Then there is an accepting state from which a second DFS starts but fails to find a cycle even though one exists.
- Let q be the first such state.
- Let r be the first flagged state that is reached from q during the second DFS and is on a cycle through q.
- Let q' be the accepting state that starts the second DFS in which r was first encountered.
- Thus, according to our assumptions, a second DFS was started from q' *before* a second DFS was started from q.

Correctness (cont.)

- Case 1: the state q' is reachable from q.
  - There is a cycle $q' \to \cdots \to r \to \cdots \to q \to \cdots \to q'$.
  - This cycle could not have been found previously; otherwise, the algorithm would have terminated.
  - This contradicts our assumption that q is the first accepting state from which the second DFS missed a cycle.
- Case 2: the state q' is not reachable from q.
  - q' cannot appear on a cycle; otherwise, q would not be the first node to start the second DFS and miss a cycle.
  - q is reachable from r and q'.
  - If q' does not occur on a cycle, by the lemma we must have backtracked from q in the first DFS before from q'.
  - This contradicts our assumption about the order of doing the second DFS.

Temporal Formula vs. Büchi Automaton



- The above Büchi automaton says that, whenever p holds at some point in time, q must hold at the same time or will hold at a later time.
  Note: the alphabet is {pq, p~q, ~pq, ~p~q}; q alone denotes any input symbol from {pq, ~pq}.
- It may not be easy to see that this indeed is the case.
- In linear temporal logic, this can easily be expressed as $\mathbf{G}(p \rightarrow \mathbf{F}q)$, which reads "always p implies eventually q".

LTL to Büchi Automata Translation

- We will study a tableau-based algorithm [GPVW] for obtaining a Büchi automaton from an LTL formula.
- The algorithm is geared towards being used in model checking in an on-the-fly fashion:
  It is possible to detect that a property does not hold by only constructing part of the model and of the automaton.
- The algorithm can also be used to check the validity of a temporal logic assertion.
- To apply the translation algorithm, we first convert the formula φ into the *negation normal form*.

Preprocessing of Formulae

Every LTL formula can be converted into the negation normal form:

- $\neg(p \wedge q) = (\neg p) \vee (\neg q)$
- $\neg(p \vee q) = (\neg p) \wedge (\neg q)$
- $\Diamond p$ (or $\mathbf{F}p$) $= \mathit{True}\ \mathcal{U}\ p$
- $\Box p$ (or $\mathbf{G}p$) $= \mathit{False}\ \mathcal{R}\ p$
- $\neg(p\ \mathcal{U}\ q) = (\neg p)\ \mathcal{R}\ (\neg q)$
- $\neg(p\ \mathcal{R}\ q) = (\neg p)\ \mathcal{U}\ (\neg q)$
- $\neg\bigcirc p$ (or $\neg\mathbf{X}p$) $= \bigcirc\neg p$

Data Structure of an Automaton Node

- *ID*: a string that identifies the node.
- *Incoming*: the incoming edges, represented by the IDs of the nodes with an outgoing edge leading to this node.
- *New*: a set of subformulae that must hold at this state and have not yet been processed.
- *Old*: the subformulae that must hold at this state and have already been processed.
- *Next*: the subformulae that must hold in all states that are immediate successors of states satisfying the formulae in *Old*.

The Algorithm: Start and Overview

- Start with a single node having a single incoming edge labeled *init* (i.e., from an initial node).
- The starting node has initially one obligation in *New*, namely φ, and *Old* and *Next* are initially empty.
- Expand the starting node (which generates new nodes) in an *DFS* manner.
- Fully processed nodes are put in a list called *Nodes*.

function $create\_graph(\varphi)$

$$
\begin{aligned}
&expand([ID \leftarrow new\_ID(),\\
&\qquad Incoming \leftarrow \{init\},\\
&\qquad Old \leftarrow \emptyset,\\
&\qquad New \leftarrow \{\varphi\},\\
&\qquad Next \leftarrow \emptyset],\\
&\qquad\qquad \emptyset);
\end{aligned}
$$

end function

The Algorithm: Node-Expansion

- Check if there are unprocessed obligations in *New* of the current node N.
- If *New* is empty, it means node N is fully processed and ready to be added to *Nodes*.
- Otherwise, a formula in *New* is selected, processed, and moved to *Old*.

function $expand(q, Nodes)$
    if $New(q) = \emptyset$ **then**
        if $\exists r \in Nodes : Old(r) = Old(q) \wedge Next(r) = Next(q)$ **then**
            $\ldots$
        else $\ldots$
    else let $\eta \in New(q)$;
        $New(q) := New(q) - \eta$;
        $\ldots$
end function

The Algorithm: Node-Expansion (cont.)

/* in **function** *expand* */
if $New(q) = \emptyset$ **then**
  if $\exists r \in Nodes : Old(r) = Old(q) \wedge Next(r) = Next(q)$ **then**
    $Incoming(r) := Incoming(r) \cup Incoming(q)$;
    return$(Nodes)$;
  else $expand([ID \leftarrow new\_ID(),$
    $Incoming \leftarrow \{ID(q)\},$
    $Old \leftarrow \emptyset,$
    $New \leftarrow Next(q),$
    $Next \leftarrow \emptyset], \quad Nodes \cup \{q\})$;
  end if
else let $\eta \in New(q)$;
  $New(q) := New(q) - \eta$;
  if $\eta \in Old(q)$ **then** $expand(q, Nodes)$;
  else ... /* cases according to the form of η */

The Algorithm: Updating the Nodes List

A fully processed current node *N* is added to *Nodes* as follows:

- If there already is a node in *Nodes* with the same obligations in both its *Old* and *Next* fields, the incoming edges of *N* are incorporated into those of the existing node.
- Otherwise, the current node *N* is added to *Nodes*.
- With the addition of node *N* in *Nodes*, a new current node is formed for its successor as follows:
  1. There is initially one edge from *N* to the new node.
  2. *New* is set initially to the *Next* field of *N*.
  3. *Old* and *Next* of the new node are initially empty.

The Algorithm: Node-Expansion (cont.)

A formula η in *New* is processed as follows:

- If η is just a literal (a proposition or the negation of a proposition), then
  - if $\neg\eta$ is in *Old*, the current node is discarded;
  - otherwise, η is added to *Old*.
- If η is not a literal, the current node can be split into two or not split, and new formulae can be added to the fields *New* and *Next*.
- The exact actions depend on the form of η.

The Algorithm: Node-Expansion (cont.)

case η **of**

$p \wedge q$: $q' := [ID \leftarrow new\_ID(),$
$Incoming \leftarrow Incoming(q),$
$Old \leftarrow Old(q) \cup \{\eta\},$
$New \leftarrow New(q) \cup \{p, q\},$
$Next \leftarrow Next(q)];$
$expand(q', Nodes);$

$p \vee q$: ...

$p\, \mathcal{U}\, q$: ...

$p\, \mathcal{R}\, q$: ...

$\bigcirc p$: ...

end case

The Algorithm: Node-Expansion (cont.)

Actions on η (that is not a literal):

- $\eta = p \wedge q$, then both p and q are added to *New*.
- $\eta = p \vee q$, then the node is split, adding p to *New* of one copy, and q to the other.
- $\eta = p\,\mathcal{U}\,q$ ($\cong q \vee (p \wedge \bigcirc(p\,\mathcal{U}\,q))$), then the node is split.
  For the first copy, p is added to *New* and $p\,\mathcal{U}\,q$ to *Next*.
  For the other copy, q is added to *New*.
- $\eta = p\,\mathcal{R}\,q$ ($\cong (p \wedge q) \vee (q \wedge \bigcirc(p\,\mathcal{R}\,q))$), similar to $\mathcal{U}$.
- $\eta = \bigcirc p$, then p is added to *Next*.

The Algorithm: Handling $\mathcal{U}$

case η **of**

$\ldots$

$$
\begin{aligned}
p\,\mathcal{U}\,q:\ & q_1 := [\mathit{ID} \leftarrow \mathit{new\_ID}(),\\
& \qquad \mathit{Incoming} \leftarrow \mathit{Incoming}(q),\\
& \qquad \mathit{Old} \leftarrow \mathit{Old}(q) \cup \{\eta\},\\
& \qquad \mathit{New} \leftarrow \mathit{New}(q) \cup \{p\},\\
& \qquad \mathit{Next} \leftarrow \mathit{Next}(q) \cup \{p\,\mathcal{U}\,q\}];\\
& q_2 := [\mathit{ID} \leftarrow \mathit{new\_ID}(),\\
& \qquad \mathit{Incoming} \leftarrow \mathit{Incoming}(q),\\
& \qquad \mathit{Old} \leftarrow \mathit{Old}(q) \cup \{\eta\},\\
& \qquad \mathit{New} \leftarrow \mathit{New}(q) \cup \{q\},\\
& \qquad \mathit{Next} \leftarrow \mathit{Next}(q)];\\
& \mathit{expand}(q_2, \mathit{expand}(q_1, \mathit{Nodes}));
\end{aligned}
$$

$\ldots$

end case

The Algorithm: Handling $\mathcal{R}$

case η **of**

$\ldots$

$$
\begin{aligned}
p\,\mathcal{R}\,q:\ q_1 := [&ID \leftarrow new\_ID(),\\
&Incoming \leftarrow Incoming(q),\\
&Old \leftarrow Old(q) \cup \{\eta\},\\
&New \leftarrow New(q) \cup \{q\},\\
&Next \leftarrow Next(q) \cup \{p\,\mathcal{R}\,q\}];\\
q_2 := [&ID \leftarrow new\_ID(),\\
&Incoming \leftarrow Incoming(q),\\
&Old \leftarrow Old(q) \cup \{\eta\},\\
&New \leftarrow New(q) \cup \{p, q\},\\
&Next \leftarrow Next(q)];\\
expand(&q_2, expand(q_1, Nodes));
\end{aligned}
$$

$\ldots$

end case

Nodes to GBA

The list of nodes in *Nodes* can now be converted into a generalized Büchi automaton $B = (\Sigma, Q, q_0, \Delta, F)$:

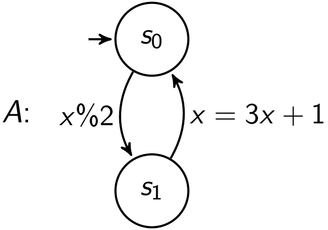
1. Σ consists of sets of propositions from AP.
2. The set of states Q includes the nodes in *Nodes* and the additional initial state q_0.
3. $(r, \alpha, r') \in \Delta$ iff $r \in$ $\mathit{Incoming}(r')$ and α satisfies the conjunction of the negated and nonnegated propositions in $\mathit{Old}(r')$
4. q_0 is the initial state, playing the role of *init*.
5. F contains a separate set F_i of states for each subformula of the form $p\,\mathcal{U}\,q$; F_i contains all the states r such that either $q \in \mathit{Old}(r)$ or $p\,\mathcal{U}\,q \notin \mathit{Old}(r)$.

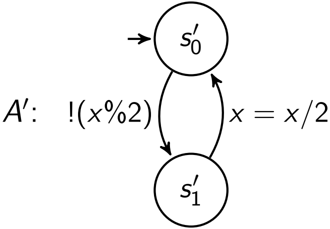
Basic Practical Details

- We now have the essential automata-based theory for model checking, but we still need to pay attention to a few more basic practical details.
- Many systems are more naturally represented as the parallel composition of several concurrently executing processes, rather than as a monolithic chunk of code.
- There are also concerns with the size of the system and the gap between the computation model and a concurrent system running on real hardware.
- Specifically, we will look into
  - asynchronous products of automata,
  - on-the-fly state exploration, and
  - fairness (in the computation model).

Processes as Automata

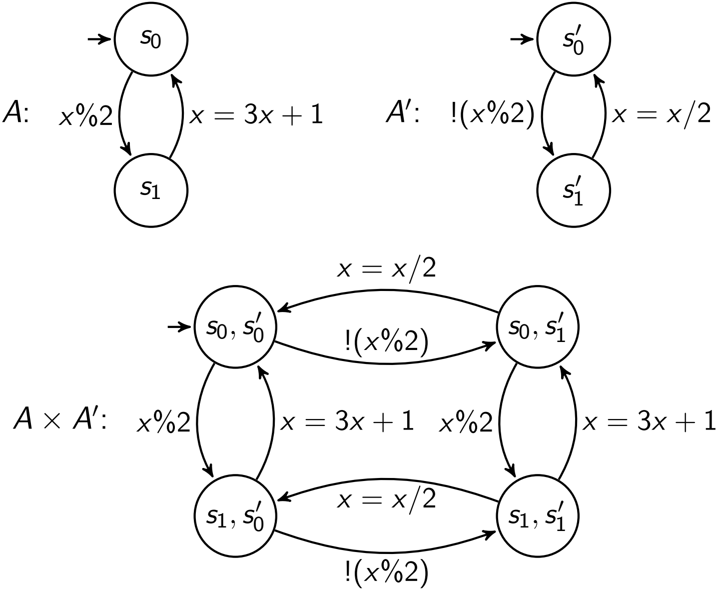
```
#define N  4
int x = N;

active proctype A0()
{
  do
  :: x%2 -> x = 3*x + 1
  od
}

active proctype A1()
{
  do
  :: !(x%2) -> x = x/2
  od
}
```





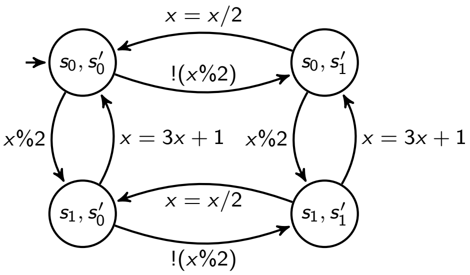
The transition labeled "$x\%2$" is enabled if $x\%2 \neq 0$, i.e., if x is odd; "$!(x\%2)$" is enabled if x is even.

Interleaving as Asynchronous Product

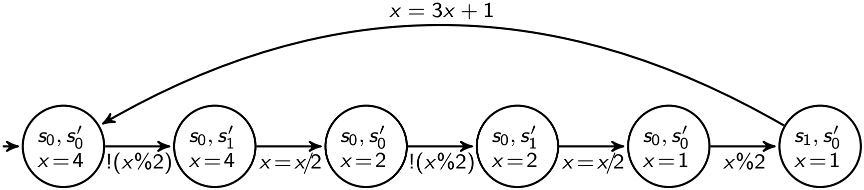


Expanded Asynchronous Product

$A0 \times A1$:

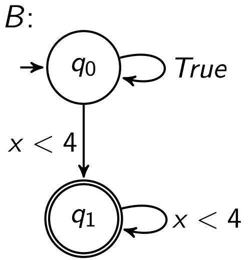


With $x = 4$ initially, we have a concrete finite-state automaton:



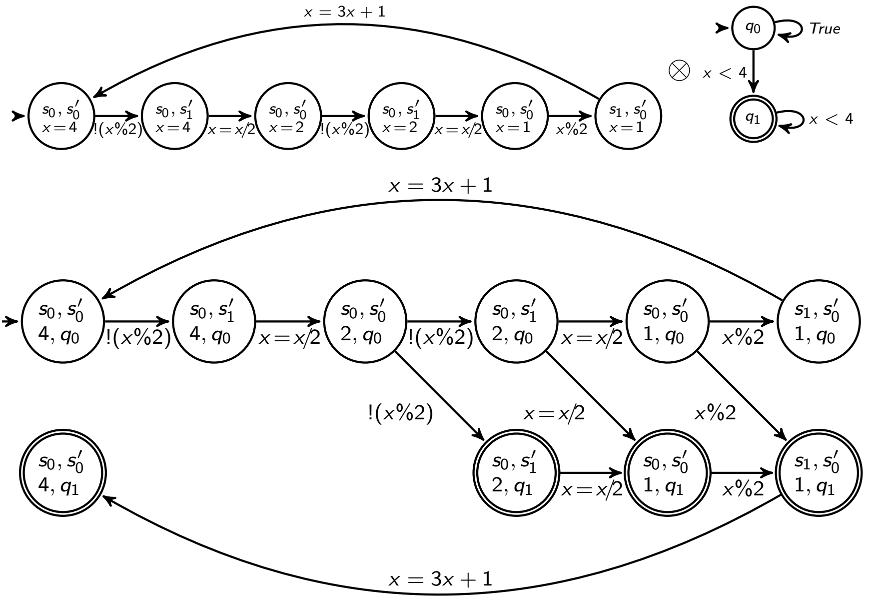
Specification as a Büchi Automaton

```
/* N was defined to be $4$ */
#define p  (x < N)

never { /* <>[]p */
T0_init:
  if
  :: p -> goto accept_S4
  :: true -> goto T0_init
  fi;
accept_S4:
  if
  :: p -> goto accept_S4
  fi;
}
```



Automaton B is equivalent to the "never claim", which specifies all the bad behaviors.

Synchronous Product



On-the-Fly State Exploration

- The automaton of the system under verification may be too large to fit into the memory.
- Using the double DFS search for a counterexample, the system (the asynchronous product automaton) need not be expanded fully.
- All we need to do are the following:
  - Keep track of the current active search path.
  - Compute the successor states of the current state.
  - Remember (by hashing) states that have been visited.
- This avoids construction of the entire system automaton and is referred to as *on-the-fly* state exploration.
- The search can stop as soon as a counterexample is found.

Fairness

IM NTU

- A concurrent system is composed of several concurrently executing processes.
- Any process that can execute a statement should eventually proceed with that instruction, reflecting the very basic fact that a normal functioning processor has a positive speed.
- This is the well-known notion of *weak fairness*, which is practically the most important kind of fairness.
- Such fairness may be enforced in one of the following two ways:
  - When searching for a counterexample, make sure that every process gets a chance to execute its next statement.
  - Encode the fairness constraint in the specification automaton.