Automata-Based Model Checking

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

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

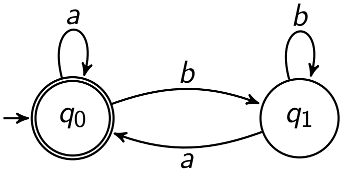
Outline

Büchi Automata

IM NTU

- 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 concurrent 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.

An Example Büchi Automaton



- A Büchi automaton accepts an infinite word if the word drives the automaton through some accepting state infinitely many times.
- The above Büchi automaton accepts infinite words over $\{a, b\}$ that have infinitely many a's.
- Using an ω-regular expression, its language is expressed as $(b^* a)^\omega$.

Büchi Automata (cont.)

- Formally, a Büchi automaton (BA), like a finite-state automaton (FA), is 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 $i \geq 0$.

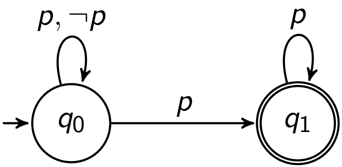
Büchi Automata (cont.)

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

$$inf(\rho) \cap F \neq \emptyset.$$

- 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.

Another Example



- This Büchi automaton has $\{p, \neg p\}$ as its alphabet.
- It accepts infinite words/sequences over $\{p, \neg p\}$ that eventually remain p forever.
- Its language corresponds to the set of sequences that satisfy the temporal formula $\Diamond\Box p$.

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 $F_i \in F$, $inf(\rho) \cap F_i \neq \emptyset$.
- 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 $L(B') = L(B)$.

The Model Checking Problem

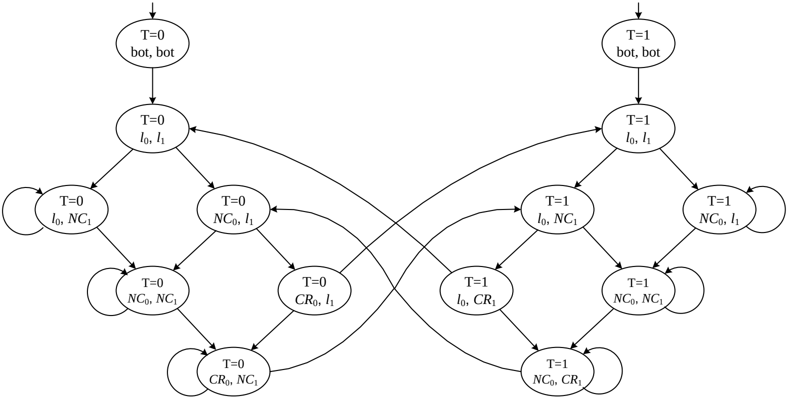
- Let AP be a set of atomic propositions.
- A Kripke structure M over AP is a 4-tuple $M = (S, R, S_0, L)$:
  1. S is a finite set of states.
  2. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
  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.
- A *computation* or *path* of M from a state s is an infinite sequence of states $\sigma = s_0, s_1, s_2, \cdots$ such that $s_0 \in S_0$ and $(s_i, s_{i+1}) \in R$, for all $i \geq 0$.
- The Model Checking problem is to determine if the computations from the initial states of a Kripke structure M satisfy a property φ expressed as a temporal formula, i.e., if $M \models \varphi$.

A Mutual Exclusion Program

$$P_{MX} = m : \textbf{cobegin } P_0 \parallel P_1 \textbf{ coend } m'$$

$$
\begin{array}{ll}
P_0 = & \\
l_0 : \textbf{while } \mathit{True} \textbf{ do} & \\
\quad NC_0 : \textbf{wait } T = 0; & \\
\quad CR_0 : T := 1; & \\
\quad \textbf{od}; & \\
l_0' &
\end{array}
\qquad
\begin{array}{ll}
P_1 = & \\
l_1 : \textbf{while } \mathit{True} \textbf{ do} & \\
\quad NC_1 : \textbf{wait } T = 1; & \\
\quad CR_1 : T := 0; & \\
\quad \textbf{od}; & \\
l_1' &
\end{array}
$$

Kripke Structure of the Program P_{MX}



The value of the outer program counter is not shown. Initially, the program counters of both processes have the value `bot` ($\perp$), indicating that they are not started yet.

Model Checking Using Automata

- 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

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

- 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.
- $B_1 \otimes B_2 =$
  $(\Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta, Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times Q_2 \times \{2\})$.
- 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:

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

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 (if one exists) of a Büchi automaton $B = (\Sigma, Q, \Delta, Q^0, F)$.
- In the context of model checking, the accepting run ρ, if found, represents a *counterexample* showing that the system does not satisfy the property.
- By definition, 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 (nontrivial) 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 q0 ∈ Q^0 do
        dfs1(q0);
    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.)

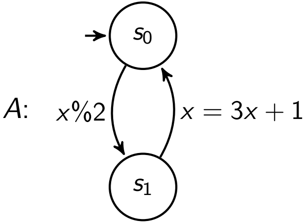
procedure *dfs2*(q)
    local q';
    flag(q);
    for all successors q' of q **do**
        if q' on *dfs1* stack **then** terminate(*False*);
        else if q' not flagged **then** *dfs2*(q');
        end if;
end procedure

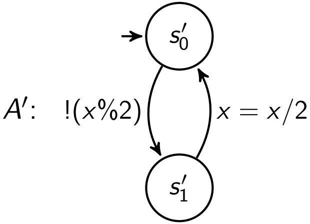
Basic Practical Details

- 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

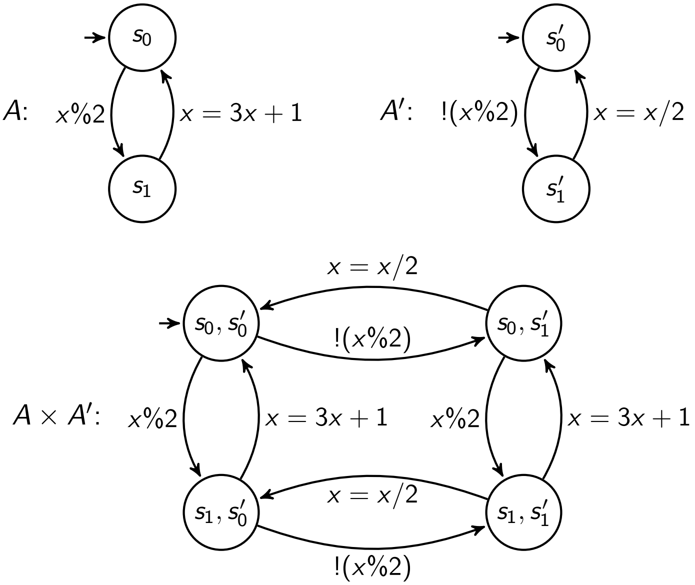
```
#define N  4
int x = N;

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

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





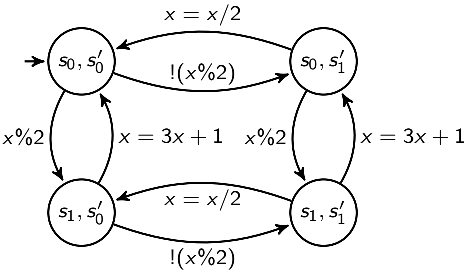
The transition labeled "$x\%2$" is enabled if $x\%2 \neq 0$, i.e., if x is odd; "$!(x\%2)$" is enabled if x is even.

Interleaving as Asynchronous Product

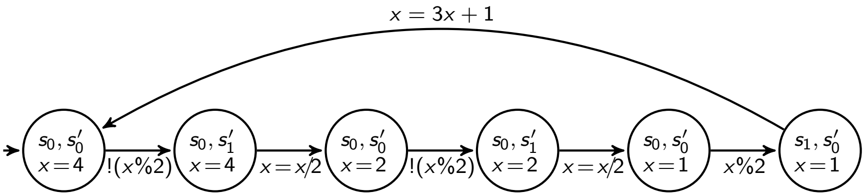


Expanded Asynchronous Product

$A0 \times A1$:



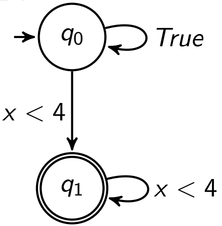
With $x = 4$ initially, we have a concrete finite-state automaton:



Specification as a Büchi Automaton

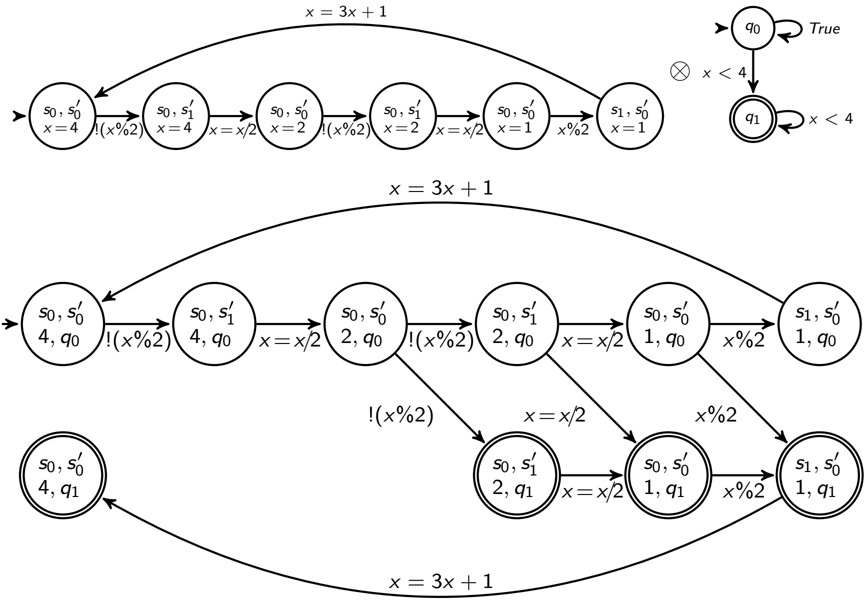
```
/* N was defined to be $4$ */
#define p  (x < N)

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

B:



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

- 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.

Concluding Remarks

- Many techniques have been developed in the past to make the automata-based approach practical for real-world applications:
  - Partial order reduction
  - Abstraction refinement
  - Compositional reasoning
- Most of these are still ongoing research.

References

- J.R. Büchi. On a decision method in restricted second-order arithmetic, in *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, Stanford University Press, 1962.
- E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*, The MIT Press, 1999.
- G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.
- W. Thomas. Automata on infinite objects, *Handbook of Theoretical Computer Science* (Vol. B), 1990.
- M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification, in LICS 1986.