Computer-Aided Verification

CS745/ECE725

Borzoo Bonakdarpour

University of Waterloo

(Fall 2013)

LTL Model Checking

Agenda

- Büchi Automata
- Linear Temporal Logic (LTL)
- Translating LTL into Büchi Automata
- The Spin Model checker

Agenda

- Büchi Automata
- Linear Temporal Logic (LTL)
- Translating LTL into Büchi Automata
- The Spin Model checker

Notation

- Σ denotes a finite *alphabet*.
- Σ^* denotes the set of *finite words* over Σ.
- Σ^ω denotes the set of *infinite words* over Σ.
- An initinite word σ is of the form $\sigma(0)\sigma(1)\cdots$ where each $\sigma(i) \in \Sigma$
- Finite words are indicated by $u, v, w, \cdots$ and the empty word by ϵ.
- Set of finite words are denoted by $U, V, W, \cdots$, and letters $\alpha, \beta, \cdots$ for ω-words.
- We use $L, L', \cdots$ to denote sets of ω-words (i.e., ω-languages).

Operators

Let W be a set of finite words:

- $\mathrm{pref} W := \{u \in \Sigma^* \mid \exists v : uv \in W\}$,
- $W^\omega := \{\alpha \in \Sigma^\omega \mid \alpha = w_0 w_1 \cdots \text{ where } w_i \in W \text{ for } i \geq 0\}$,

Let $\exists^\omega n$ mean “there exists infinitely many n”. For an ω-sequence $\sigma = \sigma(0)\sigma(1)\cdots$ in S^ω, the *infinity set* of σ is:

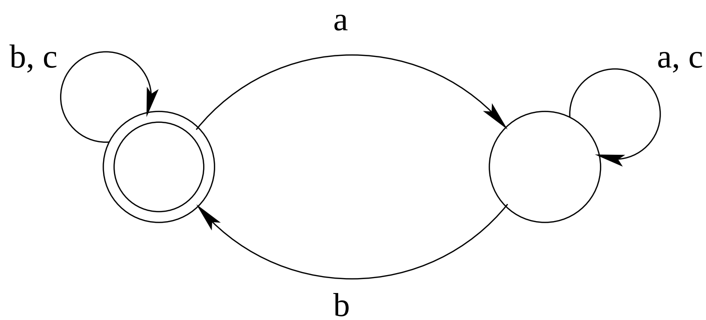
$$\mathrm{In}(\sigma) := \{s \in S \mid \exists^\omega n \sigma(n) = s\}.$$

Büchi Automata

Büchi automata are non-deterministic finite automata equipped with an *acceptance condition* that is appropriate for ω-words:

> *An ω-word is accepted if the automaton can read it from left to right while assuming a sequence of states in which some final state occurs infinitely often (Büchi Condition)*

Example



The above Büchi automaton accepts ω-words where any occurrence of letter a is followed by some occurrence of letter b.

Büchi Automata

Definition. A *Büchi automaton* over the alphabet Σ is of the form $\mathcal{A} = (Q, q_0, \Delta, F)$, where

- Q is a finite set of *states*,
- $q_0 \in Q$ is an *initial state*,
- $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition* relation, and
- $F \subseteq Q$ is a set of *final states*.

Acceptance in Büchi Automata

A *run* of $\mathcal{A}$ over an ω-word $\alpha = \alpha(0)\alpha(1)\cdots$ from Σ^ω is a sequence $\sigma = \sigma(0)\sigma(1)\cdots$ such that $\sigma(0) = q_0$ and $(\sigma(i), \alpha(i), \sigma(i+1)) \in \Delta$ for $i \geq 0$.

The run is called *successful* if $\mathrm{In}(\sigma) \cap F \neq \emptyset$.

A büchi automaton $\mathcal{A}$ *accepts* α if there a successful run of $\mathcal{A}$ on α.

Büchi Recognizable

Let

$$L(\mathcal{A}) = \{\alpha \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } \alpha\}$$

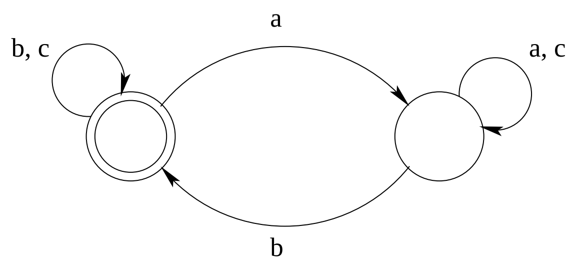
be the ω-language *recognized* by $\mathcal{A}$. If $L = L(\mathcal{A})$ for some Büchi automaton $\mathcal{A}$, L is to be *Büchi recognizable*.

Example

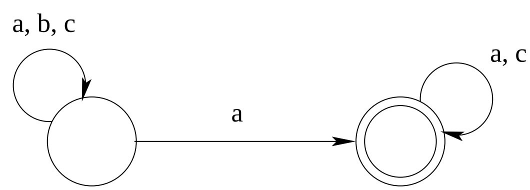
Let $\Sigma = \{a, b, c\}$. The language $L_1 \subseteq \Sigma^\omega$ defined by:

$\alpha \in L_1$ iff after any occurrence of letter a there is some occurrence of letter b in α.

A büchi automaton recognizing L_1 is the following:



The complement $\Sigma^\omega - L_1$ is recognized by the following Büchi automaton:



Main Theorems

Theorem 1. Deterministic Büchi automata are strictly less expressive than non-deterministic Büchi automata.

Theorem 2. An ω-language $L \subseteq \Sigma^\omega$ is Büchi recognizable iff L is a finite union of set $U.V^\omega$, where $U, V \subseteq \Sigma^*$ are regular sets of finite words.

Theorem 3. The emptiness problem for Büchi automata is decidable.

Theorem 4. If $L \subseteq \Sigma^\omega$ is Büchi recognizable, so is $\Sigma^\omega - L$.

Theorem 5. The inclusion problem and the equivalence problem for Büchi automata are decidable.

Agenda

- Büchi Automata
- *Linear Temporal Logic (LTL)*
- Translating LTL into Büchi Automata
- The Spin Model checker

Modal and Temporal Logic

Modal logic was originally developed by philosophers to study different *modes* of truth.

For example, the assertion P may be false in the present world, and yet the assertion *possibly* P may be true if there exists an alternate world where P is true.

Temporal logic is a special type of modal logic; it provides a formal system for qualitatively describing and reasoning about the truth values of assertions over time.

Temporal Logic

In temporal logic various temporal operators or *modalities* are provided to describe and reason about how the truth values of assertions vary with time:

- *sometimes* P which is true now if there is a future moment at which P becomes true
- *always* Q is true now if Q is true at all future moments.

Temporal Logic

Example. Two processes p_1 and p_2 request entering critical section:

- *Mutual exclusion:* always 'p_1 and p_2 do not enter the critical section simultaneously'.
- *Non-starvation:* sometime 'p_1 (resp. p_2) enters the critical section'.

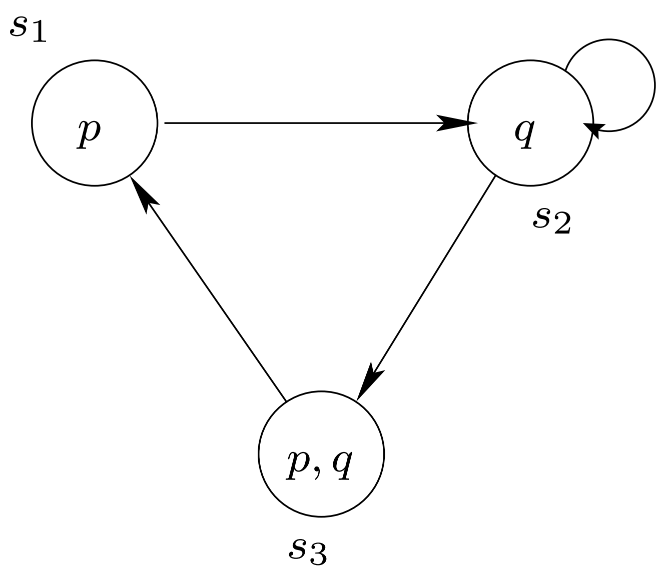
Propositional Linear Temporal Logic (LTL)

Let AP be a set of *atomic propositions*. A *Kripke structure* is $\mathcal{M} = (S, x, L)$, where

- S is a set of *states*,
- $x : \mathbb{N} \to S$ is an *infinite sequence* of states, and
- $L : S \to 2^{AP}$ is a *labelling* of each state with the set of atomic propositions in AP true at the state.

We usually employ the more convenient notation $x = (s_0, s_1, s_2, \cdots)$. We refer to x as a *path*, *computation*, or *behavior*.

Labelling



What is the labelling function in this example?

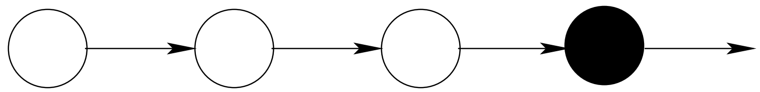
Temporal Operators

The basic temporal operators of LTL are:

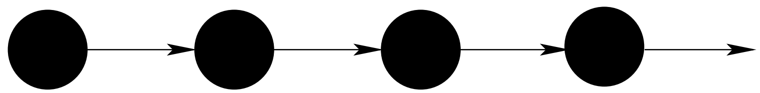
- $\Box p$: always p (also denoted Gp).
- $\Diamond p$: eventually p (also denoted Fp).
- $\bigcirc p$: nexttime p.
- p U q: p until q.

Illustration

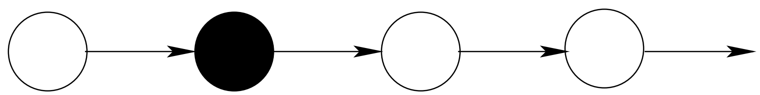
$\Diamond p$ – eventually p



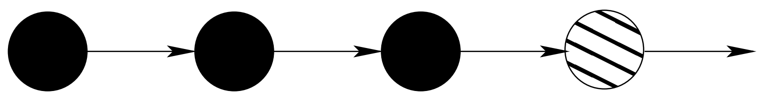
$\Box p$ – always p



$\bigcirc p$ – nexttime p



$p\mathsf{U}q$ – p until q



LTL: Syntax

The set of formulae of LTL is the least set of formulae generated by the following rules:

- each atomic proposition P is a formula
- if p and q are formulae then $p \ \wedge \ q$ and $\neg p$ are formulae
- if p and q are formulae then $p \ \mathsf{U} \ q$ and $\bigcirc p$ are formulae.

LTL: Semantics

We define the semantics of LTL with respect to a Kripke structure. We write $\mathcal{M}, x \models p$ to mean that "in structure $\mathcal{M}$ formula p is true of computation x.

Let x be a computation and x^i denote $s_i, s_{i+1}, s_{i+2} \cdots$. We define $\models$ inductively on the structure of the formulae:

1. $x \models P$ iff $P \in L(s_0)$, for atomic proposition P
2. $x \models p \wedge q$ iff $x \models p$ and $x \models q$
   $x \models \neg p$ iff it is not the case that $x \models p$
3. $x \models p$ U q iff $\exists j : (x^j \models q)$ and $\forall k < j : (x^k \models p)$,
   $x \models \bigcirc p$ iff $x^1 \models p$

LTL: Abbreviations

- $\Diamond p$ abbreviates $true$ U p
- $\Box p$ abbreviates $\neg\Diamond\neg p$.

LTL: Examples

Discuss the meaning of the following formulae:

- $\Box\Diamond p$
- $\Diamond\Box p$
- $\Box(p \Rightarrow \Diamond q)$
- $\neg(\neg p \,\mathsf{U}\, q)$

LTL Model Checking

Question. How can we check whether a Büchi automaton $\mathcal{A}$ satisfies an LTL formula ϕ (i.e., $\mathcal{A} \models \phi$)?

Answer. By checking *language inclusion*, i.e., $L(\mathcal{A}) \subseteq L(\phi)$.

Alternatively, we can check *language emptyness*; i.e., whether $L(\mathcal{A}) \cap L(\neg\phi) = \emptyset$ as follows:

1. Construct a Büchi automaton that produces all computations of $\neg\phi$ (denoted $\mathcal{A}_{\neg\phi}$)
2. Compute the product automaton $\mathcal{A}||\mathcal{A}_{\neg\phi}$
3. If $L(\mathcal{A}||\mathcal{A}_{\neg\phi}) \neq \emptyset$ then $\mathcal{A} \not\models \phi$.

LTL Model Checking

A *counterexample* is a trace of the system that violates the property. Thus, $L(\mathcal{A}||\mathcal{A}_{\neg\phi}) \neq \emptyset$ includes the set of counter examples.

An error (counterexample) may indicate a problem in the system or it may demonstrate that you did not write your property correctly.

Agenda

- Büchi Automata
- Linear Temporal Logic (LTL)
- *Translating LTL into Büchi Automata*
- The Spin Model checker

LTL to Büchi Automata

- Each state of the automata will store a set of properties that should be satisfied on paths starting at that state
  - These properties will be stored in lists *Old* and *New* where Old means already processed and New means still needs to be processed
- Each state will also store a set of properties which should be satisfied on paths starting at the next states of that state
  - These properties will be stored in the list Next
- The incoming transitions for a state will be stored in the list Incoming

LTL to Büchi Automata

- We will start with a node which has the input LTL property in its New list
- We will process the formulae in the New list of each node one by one
  - When we have $f \; \mathsf{U} \; g$ in the New list we will use $f \; \mathsf{U} \; g \equiv g \; \vee \; (f \wedge \bigcirc(f \; \mathsf{U} \; g))$

LTL to Büchi Automata

When we process a formula from a node we will either replace the node with a new node or we will replace it with two new nodes (i.e., we will split it to two nodes)

- When a node q is replaced by a node q' we will have:

$$(\mathrm{Old}(q) \wedge \mathrm{New}(q) \wedge \bigcirc\mathrm{Next}(q)) \Leftrightarrow (\mathrm{Old}(q') \wedge \mathrm{New}(q') \wedge \bigcirc\mathrm{Next}(q'))$$

- When a node q is split into two nodes q_1 and q_2 we will have

$$(\mathrm{Old}(q) \wedge \mathrm{New}(q) \wedge \bigcirc\mathrm{Next}(q)) \Leftrightarrow ((\mathrm{Old}(q_1) \wedge \mathrm{New}(q_1) \wedge \bigcirc\mathrm{Next}(q_1)) \vee (\mathrm{Old}(q_2) \wedge \mathrm{New}(q_2) \wedge \bigcirc\mathrm{Next}(q_2)))$$

Translation Algorithm

Translate(f) {

Expand([Incoming:=init, Old:=$\emptyset$, New:=f, Next:=$\emptyset$], $\emptyset$)

}

Expand(q, NodeList) {

If New(q) $=$ $\emptyset$ then

if $\exists r \in$ NodeList s.t. Old(r) = Old(q) and Next(r) = Next(q)

then Incoming(r) := Incoming(q) $\cup$ Incoming(r);

return(NodeList);

else create a new node q' s.t. Incoming(q')=q, Old(q') $= \emptyset$,

New(q')=Next(q), Next(q'):=$\emptyset$;

return expand(q', Nodelist $\cup$ $\{q\}$);

else // *New*(q) $\neq \emptyset$

pick a formula f from New(q) and remove it from New(q);

if f is already in Old(q) then return Expand(q, Nodelist);

Translation Algorithm

$$h \; \mathsf{U} \; k \equiv k \; \vee \; (h \; \wedge \; \bigcirc(h \mathsf{U} k))$$

else if $(f \equiv h \; \mathsf{U} \; k)$

create two nodes q_1 and q_2 s.t.

Incoming(q_1) = Incoming(q_2) =Incoming(q),

Old(q_1) = Old(q_2) = Old(q) $\cup$ $\{h \; \mathsf{U} \; k\}$,

New(q_1) = New(q) $\cup$ $\{h\}$,

New(q_2) = New(q) $\cup$ $\{k\}$,

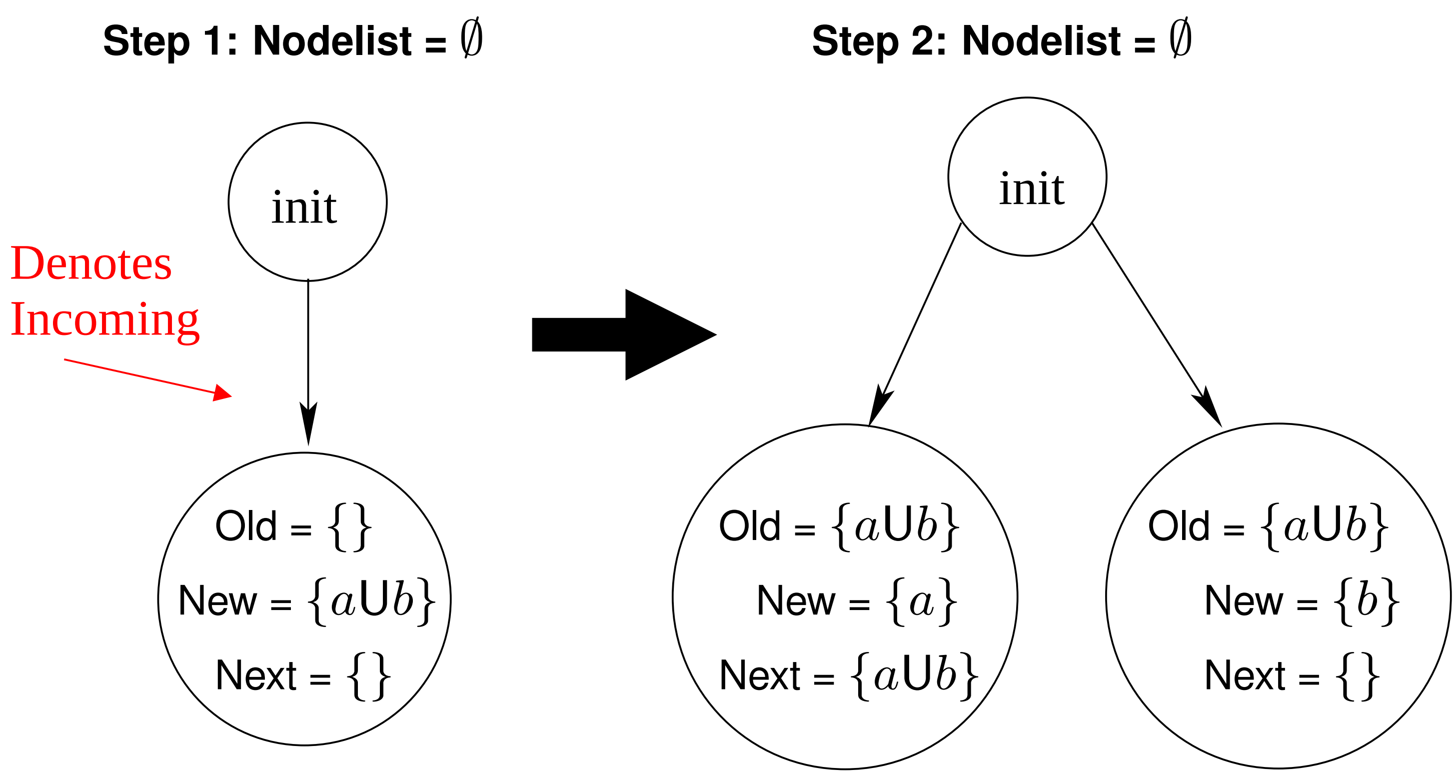
Next(q_1) = Next(q) $\cup \{h \; \mathsf{U} \; k\}$,

Next(q_2) = Next(q);

return Expand(q_1, Expand(q_2, Nodelist));

Example $(a \, \mathsf{U} \, b)$

$$a \,\mathsf{U}\, b \equiv b \vee (a \wedge \bigcirc(a \mathsf{U} b))$$



Translation Algorithm

else if ($f \in AP$ or $\neg f \in AP$ or f is a Boolean constant)

then if ($f \equiv false \ \vee \ \neg f \in$ Old(q)) then return(Nodelist);

else create a node q' s.t.

Incoming(q')=Incoming(q),

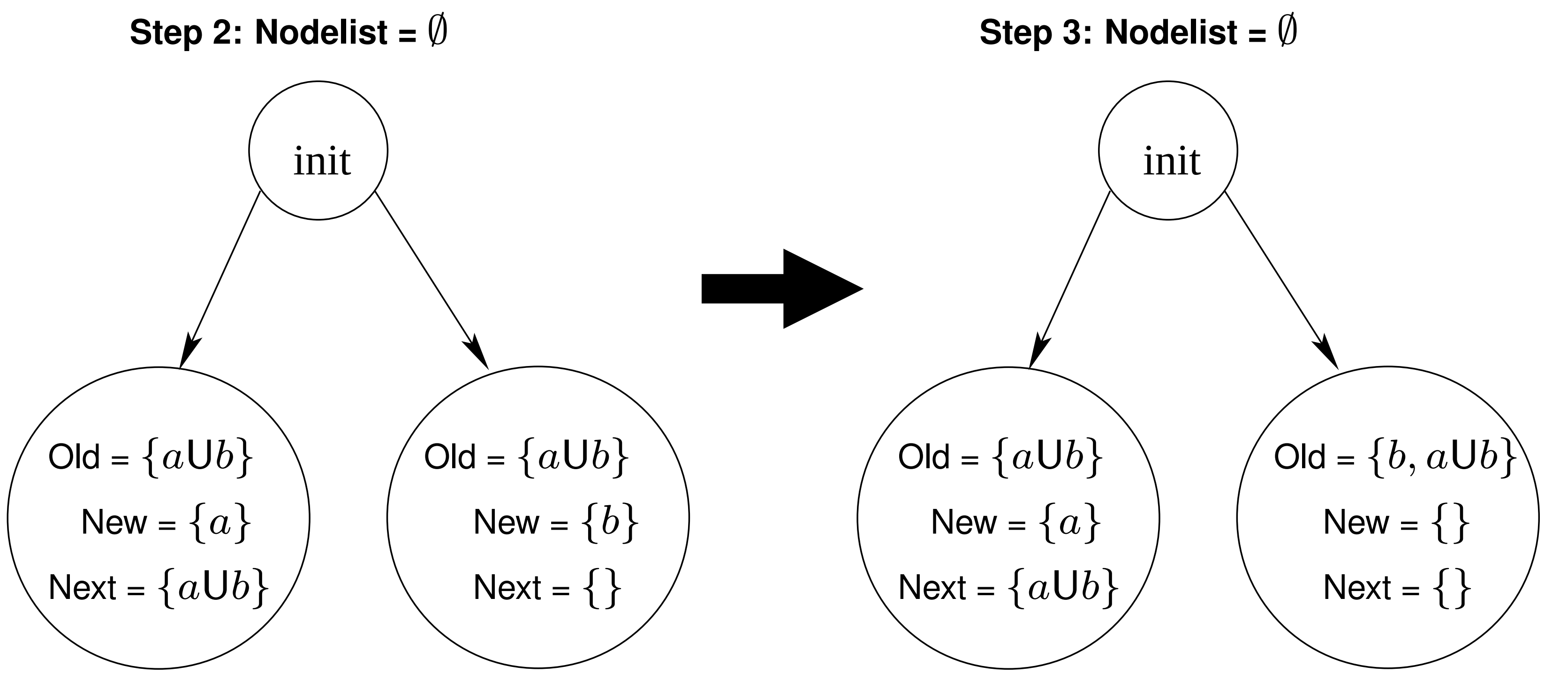
Old(q')=Old(q) $\cup \{f\}$,

New(q')=New(q) $- \{f\}$,

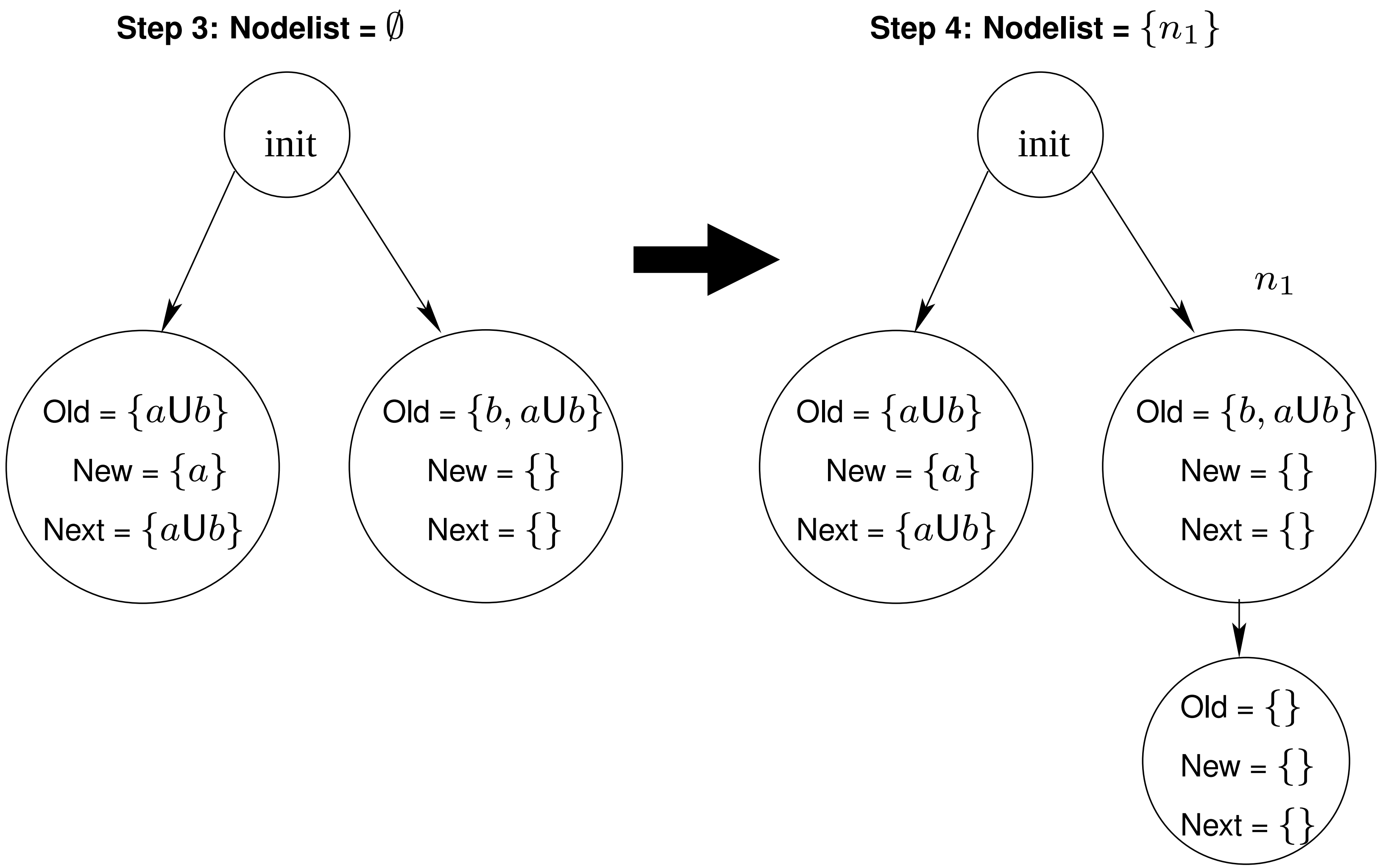
Next(q')=Next(q);

return Expand(q', Nodelist);

Example $(a \text{ U } b)$

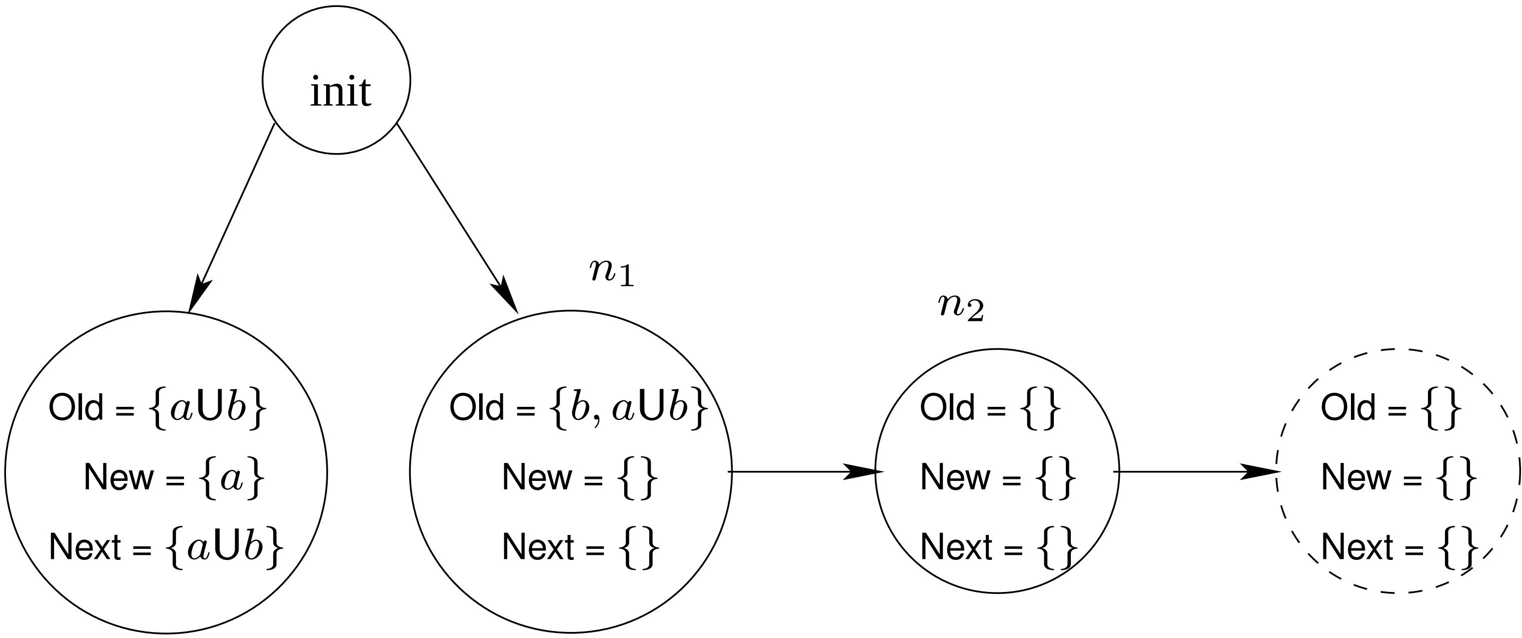


Example $(a \; \mathsf{U} \; b)$



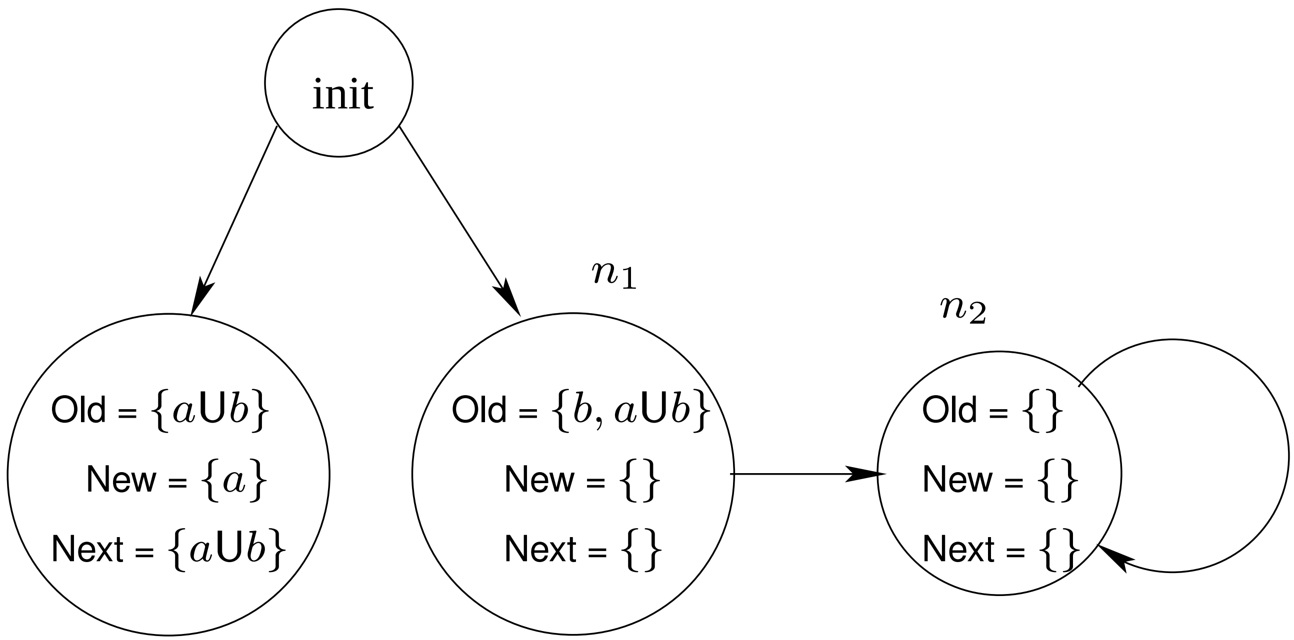
Example $(a \text{ U } b)$

Step 5: Nodelist = $\{n_1, n_2\}$



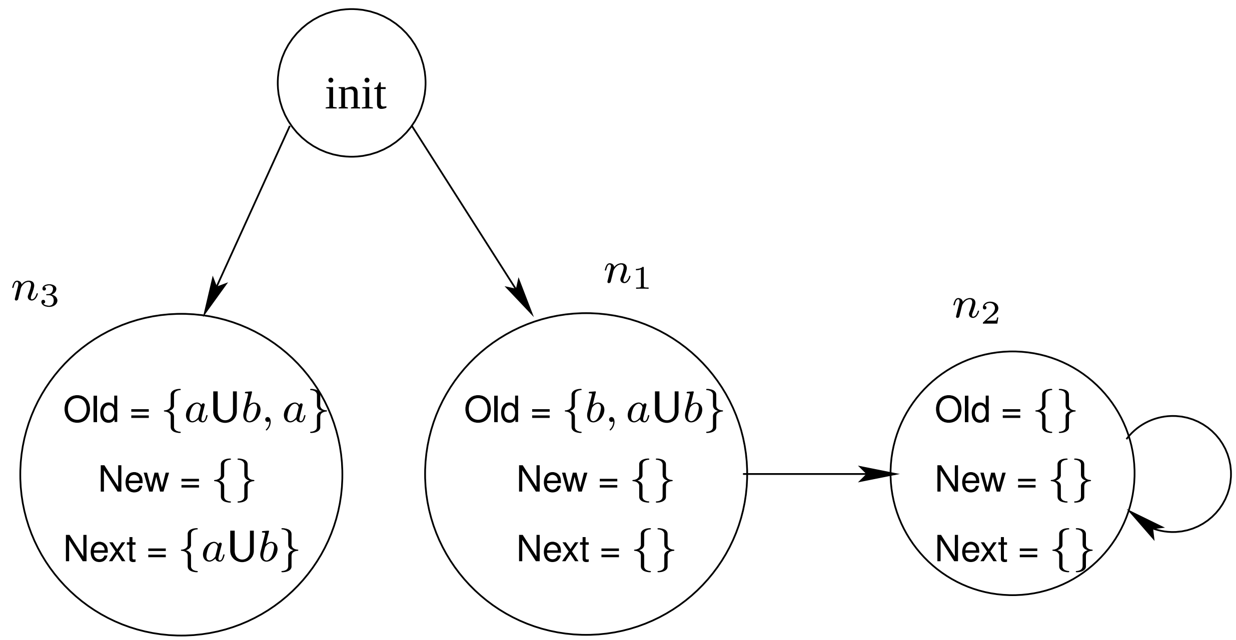
Example $(a \text{ U } b)$

Step 6: Nodelist = $\{n_1, n_2\}$



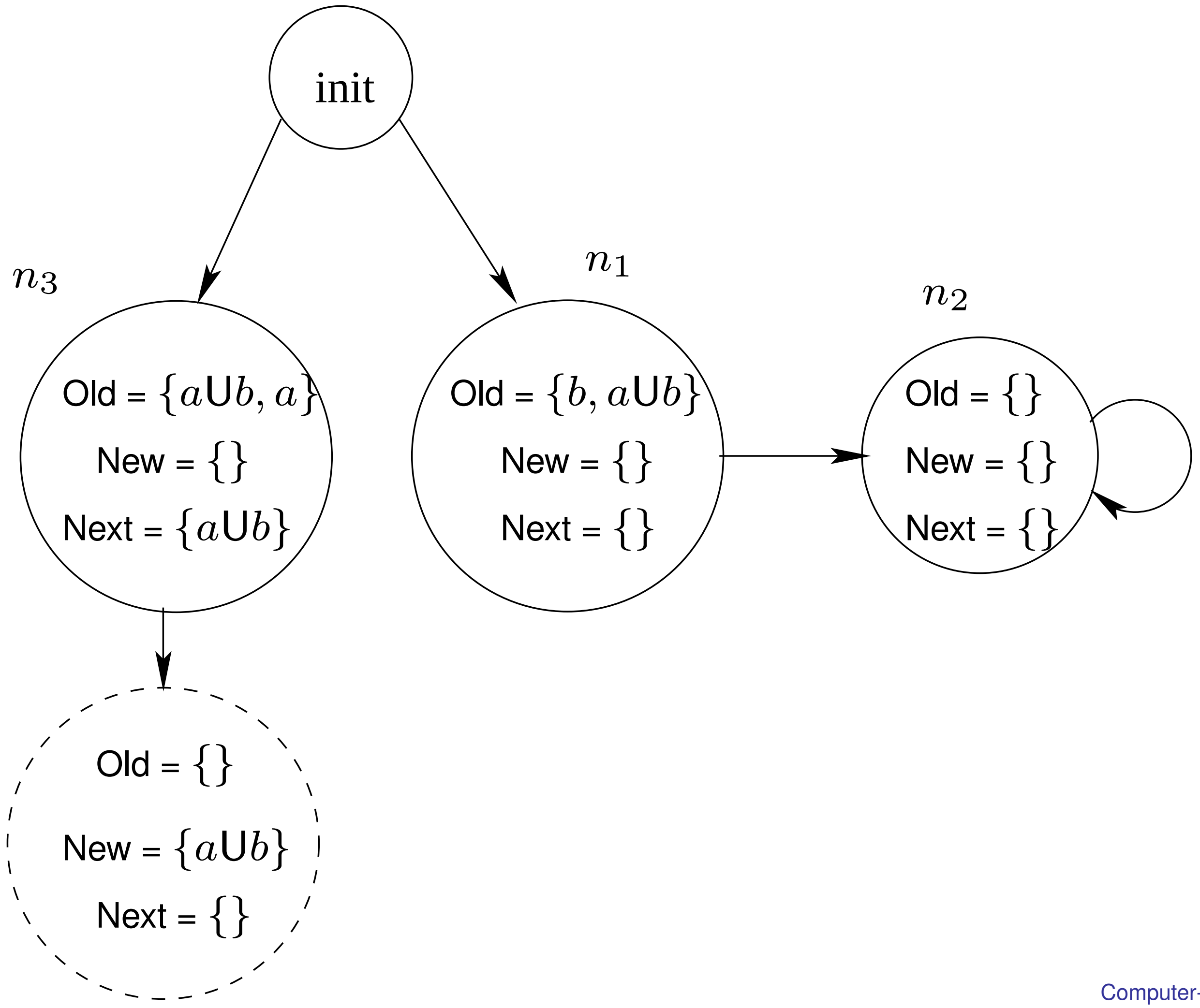
Example $(a \text{ U } b)$

Step 7: Nodelist = $\{n_1, n_2, n_3\}$



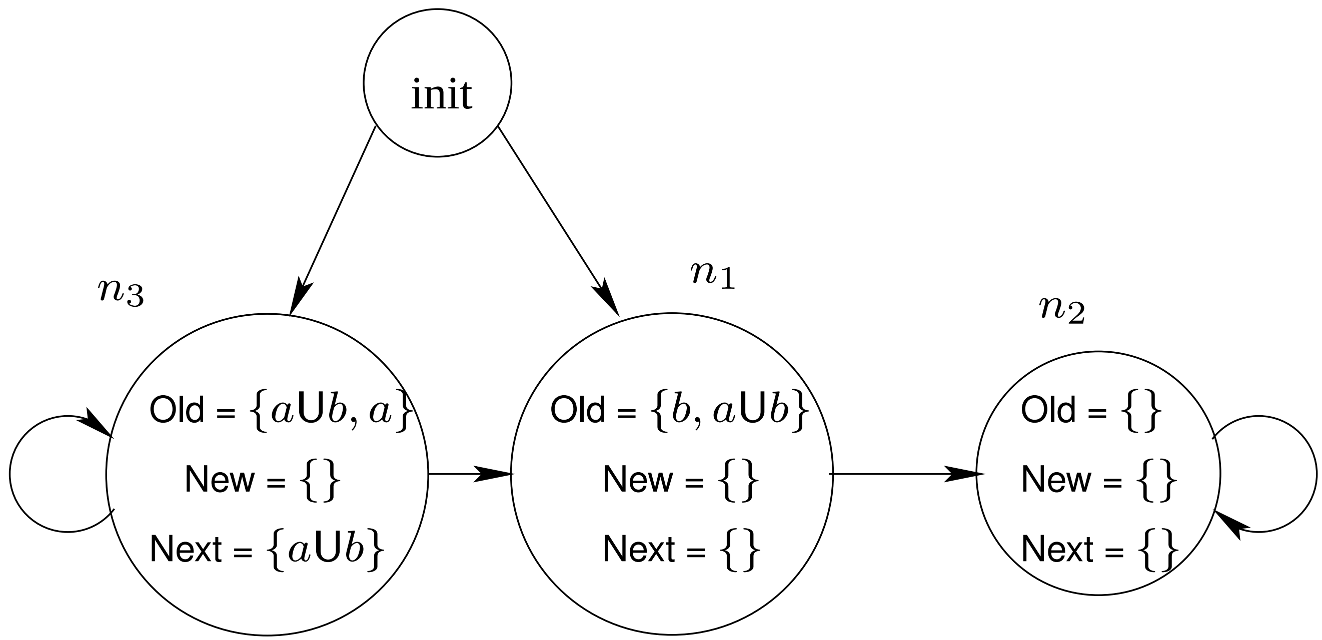
Example $(a \text{ U } b)$

Step 8: Nodelist = $\{n_1, n_2, n_3\}$



Example $(a \text{ U } b)$

Step 9: Nodelist = $\{n_1, n_2, n_3\}$



Translation Algorithm

else if $(f \equiv h \ \vee \ k)$

create two nodes q_1 and q_2 s.t

Incoming(q_1) = Incoming(q_2) = Incoming(q),

Old(q_1) = Old(q_2) = Old(q) $\cup \{h \vee k\}$,

New(q_1) = (New(q) $- \{h \vee k\}) \cup \{h\}$,

New(q_2) = (New(q) $- \{h \vee k\}) \cup \{k\}$,

Next(q_1) = Next(q_2) = Next(q);

return Expand(q_2, Expand(q_1, Nodelist));

Translation Algorithm

else if $(f \equiv h \wedge k)$

create two node q' s.t

Incoming(q') = Incoming(q),

Old(q') = Old(q) $\cup \{h \wedge k\}$,

New(q') = (New(q) $-\{h \wedge k\}) \cup \{h\} \cup \{k\}$,

Next(q') = Next(q);

return Expand(q', Nodelist);

Translation Algorithm

else if $(f \equiv \bigcirc h)$

create two node q' s.t

Incoming(q') = Incoming(q),

Old(q') = Old(q) $\cup$ $\{\bigcirc h\}$,

New(q') = (New(q) $-\{\bigcirc h\}$),

Next(q') = Next(q) $\cup$ $\{h\}$;

return Expand(q', Nodelist);

Completing the Automaton

The resulting Büchi automaton $\mathcal{A} = (Q, q_0, \Delta, F)$:

- $\Sigma = 2^{AP}$
- $Q = \mathrm{Nodelist} \ \cup \ \mathrm{init}$
- $q_0 = \mathrm{init}$
- Δ is defined as follows:
  $(q, d, q') \in \Delta$ iff $q \in \mathrm{Incoming}(q')$ and
  d satisfies the conjunction of negated and unnegated propositions in Old(q')
- $F \subseteq 2^Q$ i.e., $F = \{F_1, F_2, \cdots, F_k\}$

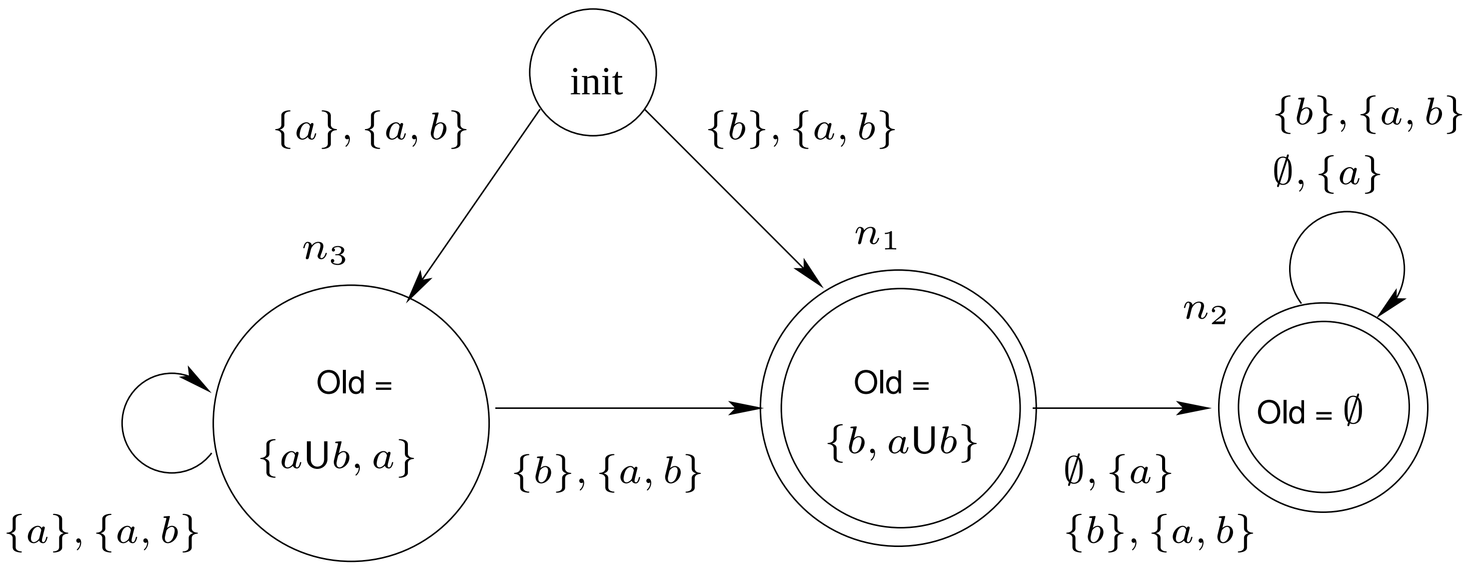
The acceptance set F contains a set of accepting states $F_i \in F$ for each subformula of the form h U k where F_i contains all the states q s.t. either $k \in \mathrm{Old}(q)$ or hU$k \notin \mathrm{Old}(q)$. If there are no subformulas of the form hUk then $F = \{Q\}$

Completing the Automaton

The size of the resulting automaton can be *exponential* in the size of the input formula

The resulting automaton is a generalized Büchi automaton we can translate it to a standard Büchi automaton.

Example $(a \,\mathsf{U}\, b)$



$\Sigma = 2^{AP} = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}$

$F = \{\{n_1, n_2\}\}$

$Q = \{\text{init}, n_1, n_2, n_3\}$

$q_0 = \text{init}$

Checking Emptyness

Let $\mathcal{A}$ be a Büchi automaton. Recall that:

$$L(\mathcal{A}) = \{\alpha \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } \alpha\}$$

$L(\mathcal{A})$ is nonempty if there exists an accepting state $q \in F$ such that:

- q is reachable from initial state in q_0, and
- q is reachable from itself (i.e., q is contained in a cycle).

Checking Emptyness

Any run of a Büchi automaton has a suffix in which all the states on that suffix appear infinitely many times:

- Each state on that suffix is reachable from any other state
- Hence these states form a *strongly connected component*
- If there is an accepting state among those states than the run is an accepting run

So emptiness check involves finding a *strongly connected component* that contains an accepting state and is reachable from an initial state

Checking Emptyness

To find cycles in a graph one can use a *depth-first search algorithm* which constructs the strongly connected components in linear time by adding two integer numbers to every state reached.

If a strongly connected component reachable from an initial state contains an accepting state then the language accepted by the Büchi automaton is not empty.

There is a more memory efficient algorithm for checking the same condition which is called *nested depth first search*.

Agenda

- Büchi Automata
- Linear Temporal Logic (LTL)
- Translating LTL into Büchi Automata
- *The Spin Model checker*

Spin

- Model-checker
- Based on automata theory
- Allows LTL or automata specification
- Efficient (on-the-fly model checking, partial order reduction).
- Developed in Bell Laboratories.

The Language of Spin (Promela)

- The expressions are from C.
- The communication is from CSP.
- The constructs are from Guarded Command.

Expressions

- Arithmetic: `+, -, *, /, %`
- Comparison: `>, >=, <, <=, ==, !=`
- Boolean: `&&, ||, !`
- Assignment: `:=`
- Increment/decrement: `++, --`

Expressions

- byte name1, name2=4, name3;
- bit b1,b2,b3;
- short s1,s2;
- int arr1[5];

Message types and channels

- mtype = {OK, READY, ACK}
- mtype Mvar = ACK
- chan Ng=[2] of {byte, byte, mtype}, Next=[0] of {byte}

  Ng has a buffer of 2, each message consists of two bytes and an enumerable type (mtype). Next is used with handshake message passing.

Sending and receiving a message

- *Channel declaration:*
  chan qname=[3] of mtype, byte, byte
- *In sender:*
  qname!tag3(expr1, expr2) or equivalently:
  qname!tag3, expr1, expr2
- *In Receiver:*
  qname?tag3(var1,var2)

Condition

```
if
:: x%2==1 -> z=z*y; x–
:: x%2==0 -> y=y*y; x=x/2
fi
```

If more than one guard is enabled: a non-deterministic choice.

If no guard is enabled: the process waits (until a guard becomes enabled).

Looping

```
do
:: x>y -> x=x-y
:: y>x -> y=y-x
:: else break
od;
```

Normal way to terminate a loop: with break. (or goto).

As in condition, we may have a non-deterministic loop or have to wait.

Process Declaration

Definition of a process:

```
proctype prname (byte Id; chan Comm)
{
        statements
}
```

Activation of a process:

```
run prname (7, Con[1]);
```

init process is the root of all others

init{ statements }

```
init {byte I=0;
        atomic{do
                ::I<10 -> run prname(I, chan[I]);
                        I=I+1
                ::I=10 -> break;
                od}}
```

atomic allows performing several actions as one atomic step.

Mutual Exclusion

```
loop
        Non_Critical_Section;
        TR:Pre_Protocol;
        CR:Critical_Section;
        Post_protocol;
end loop;
```

Mutual Exclusion

```
task P0 is
begin
    loop
        Non_Critical_Sec;
        Wait Turn=0;
        Critical_Sec;
        Turn:=1;
    end loop
end P0.
```

```
task P1 is
begin
    loop
        Non_Critical_Sec;
        Wait Turn=1;
        Critical_Sec;
        Turn :=0;
    end loop
end P1
```

Translating into Spin

```
#define t (P@try)
#define c (P@cr)
#define critical (incrit[0] && incrit[1])
byte turn=0, incrit[2]=0;
proctype P (bool id)
{ do
     :: 1 ->
        do
           :: 1 -> skip
           :: 1 -> break
        od
try:do
      ::turn==id -> break
      od;
cr:incrit[id]=1;
     incrit[id]=0;
     turn=1-turn
     od}
init{ atomic{
     run P(0); run P(1) } };
```

LTL Verification Using Spin

Both process do not enter the critical section:

spin -f '[] !critical'

spin -f '[](t -> <>c)'

In old versions of Spin, one could verify properties expressed as *never claims*.