INF5140 – Specification and Verification of Parallel Systems

Lecture 5 - Introduction to Logical Model Checking and Theoretical Foundations

Spring 2017

Institutt for informatikk, Universitetet i Oslo

March 3, 2017

Credits:

- Many slides (all the figures with blue background and few others) were taken from Holzmann's slides on "Logical Model Checking", course given at Caltech http://spinroot.com/spin/Doc/course/index.html (http://spinroot.com/spin/Doc/course/index.html)

Outline

2. Logic Model Checking: What is it about?
    - The Basic Method
    - General remarks
    - Motivating Examples
3. Automata and Logic
    - Finite State Automata
    - Büchi Automata
    - Something on Logic and Automata
    - Implications in Model Checking
    - Automata Products
4. Model Checking Algorithm
    - Preliminaries
    - The Algorithm
5. Final Remarks
    - Something on Automata

Logic Model Checking: What is it about?

Logic Model Checking (1)

- Model checking is a technique for verifying *finite-state* concurrent systems
- Theoretically speaking, model checking consists of the following tasks:

1. Modeling the system
    - It may require the use of abstraction
    - Often using some kind of automaton
2. Specifying the properties the design must satisfy
    - It is impossible to determine all the properties the systems should satisfy
    - Often using some kind of temporal logic
3. Verifying that the system satisfies its specification
    - In case of a negative result: error trace
    - An error trace may be product of a specification error

Logic Model Checking (2)

The *application* of model checking at the design stage of a system typically consists of the following steps:

1. Choose the properties (correctness requirements) critical to the sytem you want to build (software, hardware, protocols)
2. Build a model of the system (will use for verification) guided by the above correctness requirements
   - The model should be as small as possible (for efficiency)
   - It should, however, capture everything which is relevant to the properties to be verified
3. Select the appropriate verification method based on the model and the properties (LTL-, CTL*-based, probabilistic, timed, weighted)
4. Refine the verification model and correctness requirements until all correctness concerns are adequately satisfied

- Main causes of combinatorial complexity in SPIN/Promela[3]
  - The number of and size of buffered channels
  - The number of asynchronous processes

[3] and in other model checkers.

Logic Model Checking (2)

The *application* of model checking at the design stage of a system typically consists of the following steps:

1. Choose the properties (correctness requirements) critical to the sytem you want to build (software, hardware, protocols)
2. Build a model of the system (will use for verification) guided by the above correctness requirements
   - The model should be as small as possible (for efficiency)
   - It should, however, capture everything which is relevant to the properties to be verified
3. Select the appropriate verification method based on the model and the properties (LTL-, CTL*-based, probabilistic, timed, weighted)
4. Refine the verification model and correctness requirements until all correctness concerns are adequately satisfied

- Main causes of combinatorial complexity in SPIN/Promela[3]
  - The number of and size of buffered channels
  - The number of asynchronous processes

[3] and in other model checkers.

The Basic Method

There are different model checking techniques. We will use the *automata-theoretic approach* which is implemented in the SPIN model checker (tool). Theoretically:

- System: $\mathcal{L}(S)$ (set of possible behaviors/traces/words of S)
- Property: $\mathcal{L}(P)$ (the set of valid/desirable behaviors)
- Prove that $\mathcal{L}(S) \subseteq \mathcal{L}(P)$ (everything possible is valid)
  - Proving language inclusion is complicated
- Method
  - Let $\overline{\mathcal{L}(P)}$ be the language $\Sigma^\omega \setminus \mathcal{L}(P)$ of words not accepted by P
  - Prove $\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset$
    - There is no accepted word by S disallowed by P

This will be clear at the end of the talk, ... I hope

The Basic Method

There are different model checking techniques. We will use the *automata-theoretic approach* which is implemented in the SPIN model checker (tool). Theoretically:

- System: $\mathcal{L}(S)$ (set of possible behaviors/traces/words of S)
- Property: $\mathcal{L}(P)$ (the set of valid/desirable behaviors)
- Prove that $\mathcal{L}(S) \subseteq \mathcal{L}(P)$ (everything possible is valid)
  - Proving language inclusion is complicated
- Method
  - Let $\overline{\mathcal{L}(P)}$ be the language $\Sigma^{\omega} \setminus \mathcal{L}(P)$ of words not accepted by P
  - Prove $\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset$
    - There is no accepted word by S disallowed by P

This will be clear at the end of the talk, ... I hope

The Basic Method

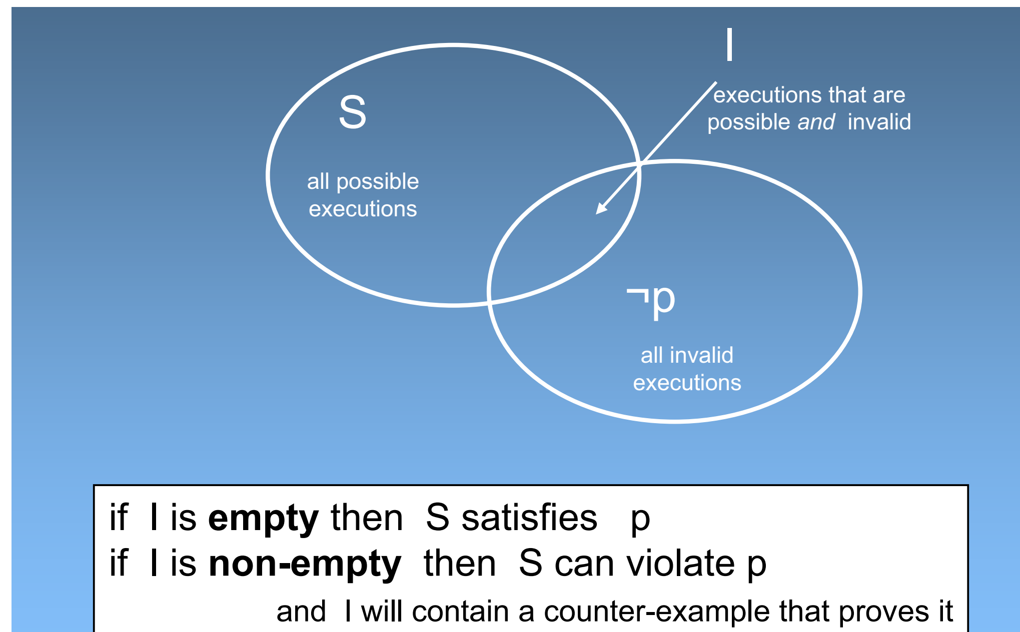
There are different model checking techniques. We will use the *automata-theoretic approach* which is implemented in the SPIN model checker (tool). Theoretically:

- System: $\mathcal{L}(S)$ (set of possible behaviors/traces/words of S)
- Property: $\mathcal{L}(P)$ (the set of valid/desirable behaviors)
- Prove that $\mathcal{L}(S) \subseteq \mathcal{L}(P)$ (everything possible is valid)
  - Proving language inclusion is complicated
- Method
  - Let $\overline{\mathcal{L}(P)}$ be the language $\Sigma^\omega \setminus \mathcal{L}(P)$ of words not accepted by P
  - Prove $\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset$
    - There is no accepted word by S disallowed by P

This will be clear at the end of the talk, ... I hope

The Basic Method

Graphically:

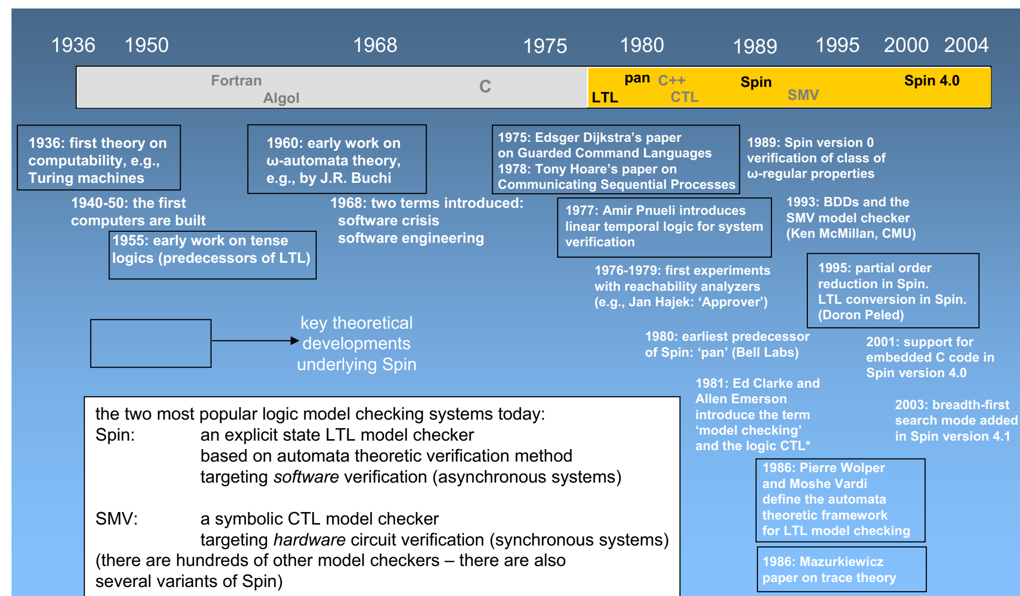


Scope of the method

- Logic model checkers (LMC) are suitable for concurrent and multi-threading finite state systems
- Some of the errors LMC may catch:
  - Deadlocks (two or more competing processes are waiting for the other to finish, and thus neither ever does)
  - Livelocks (two or more processes continually change their state in response to changes in the other processes)
  - Starvation (a process is perpetually denied access to necessary resources)
  - Priority and locking problems
  - Race conditions (attempting to perform two or more operations at the same time, which must be done in the proper sequence in order to be done correctly)
  - Resource allocation problems
  - Dead code (unreachable code)
  - Violation of certain system bounds
  - Logic problems: e.g, temporal relations

A bit of history

The following diagram shows the evolution of the theoretical foundations of LMC:



On correctness (reminder)

- A system is correct if it meets its design requirements.
  - There is no notion of "absolute" correctness: It is always w.r.t. a given specification
- Getting the properties (requirements) right is as important as getting the model of the system right
- Examples of correctness requirements
  - A system should not deadlock
  - No process should starve another
  - Fairness assumptions
    - E.g., an infinite often enabled process should be executed infinitely often
  - Causal relations
    - E.g., each time a request is send, and acknowledgment must be received (*response* property)

On correctness (reminder)

- A system is correct if it meets its design requirements.
  - There is no notion of "absolute" correctness: It is always w.r.t. a given specification
- Getting the properties (requirements) right is as important as getting the model of the system right
- Examples of correctness requirements
  - A system should not deadlock
  - No process should starve another
  - Fairness assumptions
    - E.g., an infinite often enabled process should be executed infinitely often
  - Causal relations
    - E.g., each time a request is send, and acknowledgment must be received (*response* property)

On models and abstraction

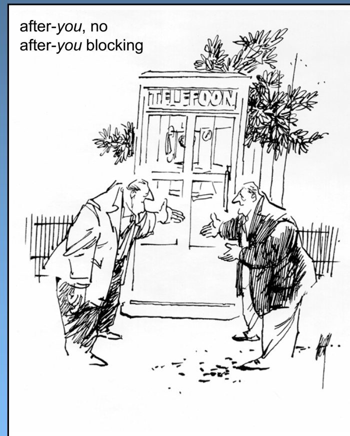
- The use of abstraction is needed for building models (systems may be extremely big)
  - A model is always an abstraction of the reality
- The choice of the model/abstractions depends on the requirements to be checked
- A good model keeps only relevant information
  - A trade-off must be found: too much detail may complicate the model; too much abstraction may oversimplify the reality
- Time and probability are usually abstracted away in LMC

Building verification models

- Statements about system design and system requirement must be separated
  - One formalism for specifying behavior (system design)
  - Another formalism for specifying system requirements (correctness properties)
- The two types of statements define a verification model
- A model checker can now
  - Check that the behavior specification (the design) is logically consistent with the requirement specification (the desired properties)

Two asynchronous processes may easily get blocked when competing for a shared resource

in real-life conflicts ultimately get resolved by *human judgment.*
computers, though, must be able to resolve it with fixed algorithms



A Small Multi-threaded Program

```
int   x,  y,  r;
int  *p, *q, *z;
int **a;

thread_1(void)      /* initialize p, q, and r */
{
    p = &x;
    q = &y;
    z = &r;
}
thread_2(void)      /* swap contents of x and y */
{
    r = *p;
    *p = *q;
    *q = r;
}
thread_3(void)      /* access z via a and p */
{
    a = &p;
    *a = z;
    **a = 12;
}
```

3 asynchronous threads
accessing shared data
3 statements each
how many test runs are needed to
check that no data corruption can occur?

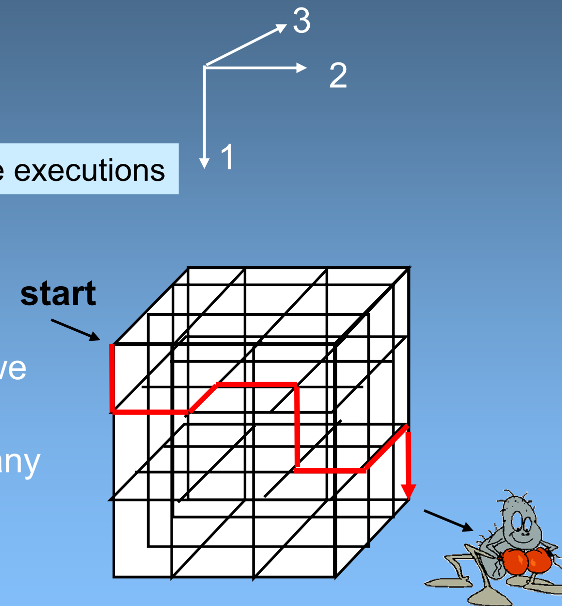
Thread Interleaving

- the number of possible thread interleavings is...

$$\frac{9!}{6!.3!} \cdot \frac{6!}{3!.3!} \cdot \frac{3!}{3!} =$$ 1,680 possible executions

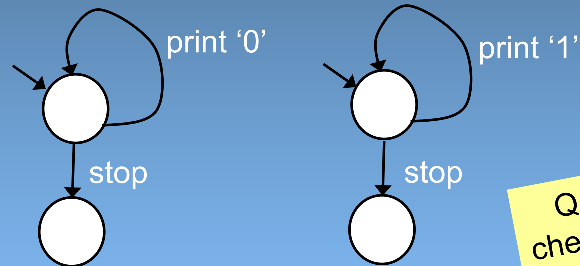
placing 3 sets of 3 tokens in 9 slots

- are all these executions okay?
- can we check them all? should we check them all?
- in classic system testing, how many would normally be checked?



A Simpler Example

- consider two 2-state automata
  - representing two asynchronous processes
- one can print an arbitrary number of ‘0’ digits, or stop
- the other can print an arbitrary number of ‘1’ digits, or stop



Q: how could a model checker deal with possibly infinite executions?

how many different combined executions are there?
i.e., how many different binary numbers can be printed?
how would one test that this system does what we think it does?

Automata and Logic

Finite State Automata

Definition

A finite state automaton is a tuple (S, s_0, L, F, T), where

- S is finite set of states
- $s_0 \in S$ is a distinguished initial state
- L is a finite set of labels (symbols)
- $F \subseteq S$ is the (possibly empty) set of final states
- $T \subseteq S \times L \times S$ is the transition relation, connecting states in S

We will, in general, follow Holzmann's notation: $A.S$ denotes the state S of automaton A, $A.T$ denotes the transition relation T of A, and so on....

If understood from the context, we will avoid the use of A.

Finite State Automata

Definition

A finite state automaton is a tuple (S, s_0, L, F, T), where

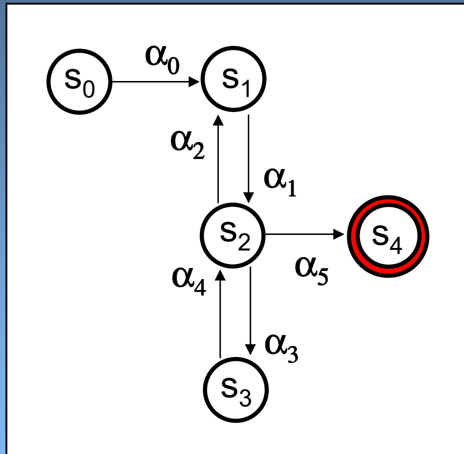
- S is finite set of states
- $s_0 \in S$ is a distinguished initial state
- L is a finite set of labels (symbols)
- $F \subseteq S$ is the (possibly empty) set of final states
- $T \subseteq S \times L \times S$ is the transition relation, connecting states in S

We will, in general, follow Holzmann's notation: $A.S$ denotes the state S of automaton A, $A.T$ denotes the transition relation T of A, and so on....

If understood from the context, we will avoid the use of $A._$

Finite State Automata
Example

A: {S, s0, L, F, T}



A.S: { s_0, s_1, s_2, s_3, s_4 }

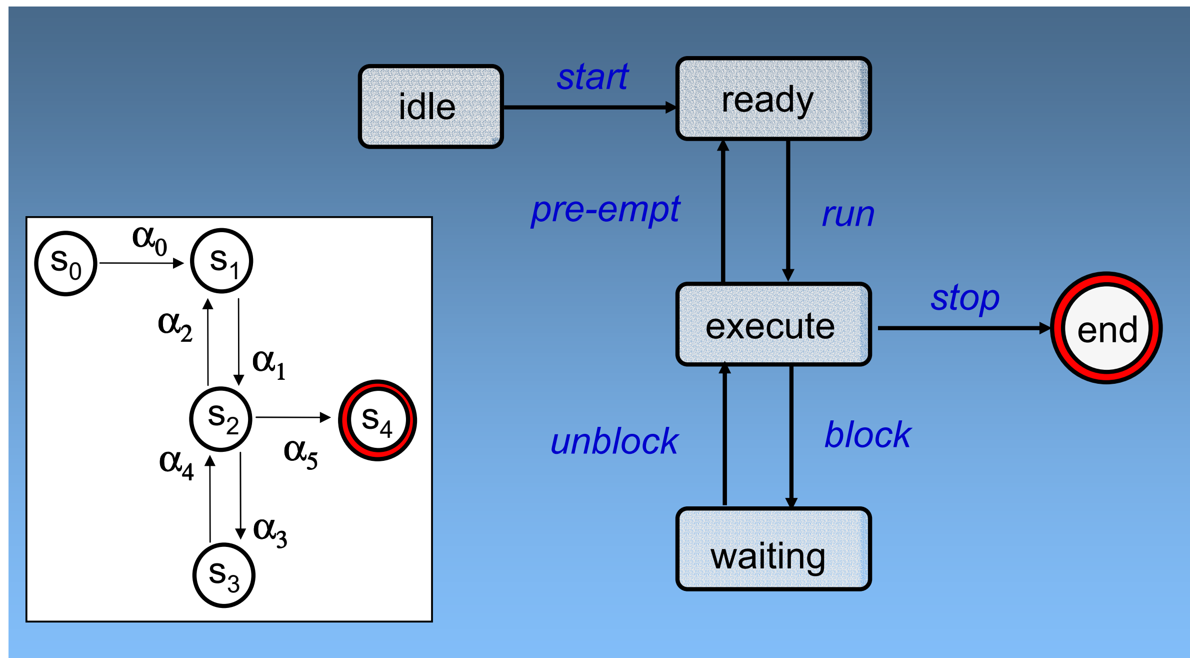
A.L = { α_0, α_1, α_2, α_3, α_4, α_5 }

A.F = { s_4 }

A.T = {(s_0,α_0,s_1), (s_1, α_1,s_2), ...}

Finite State Automata
Example: An Interpretation

The above automaton may be interpreted as a Process Scheduler:



Determinism vs. non-determinism

Definition

A finite state automaton $A = (S, s_0, L, F, T)$ is deterministic iff

$$\forall s\ \forall l, ((s, l, s') \in A.T \wedge (s, l, s'') \in A.T) \implies s' \equiv s''$$

I.e., the destination state of a transition is uniquely determined by the source state and the transition label. An automaton is called non-deterministic if it does not have this property

Examples:

- The automaton corresponding to the process scheduler is deterministic
- Automaton from definition is non-deterministic
(think of distinction between *relations* and *partial functions*)

Determinism vs. non-determinism

Definition

A finite state automaton $A = (S, s_0, L, F, T)$ is deterministic iff

$$\forall s\ \forall l, ((s, l, s') \in A.T \wedge (s, l, s'') \in A.T) \implies s' \equiv s''$$

I.e., the destination state of a transition is uniquely determined by the source state and the transition label. An automaton is called non-deterministic if it does not have this property

Examples:

- The automaton corresponding to the process scheduler is deterministic
- Automaton from definition is non-deterministic (think of distinction between *relations* and *partial functions*)

Definition of a Run

Definition

A run of a finite state automaton $A = (S, s_0, L, F, T)$ is an ordered and possibly infinite set of transitions (a sequence) from T

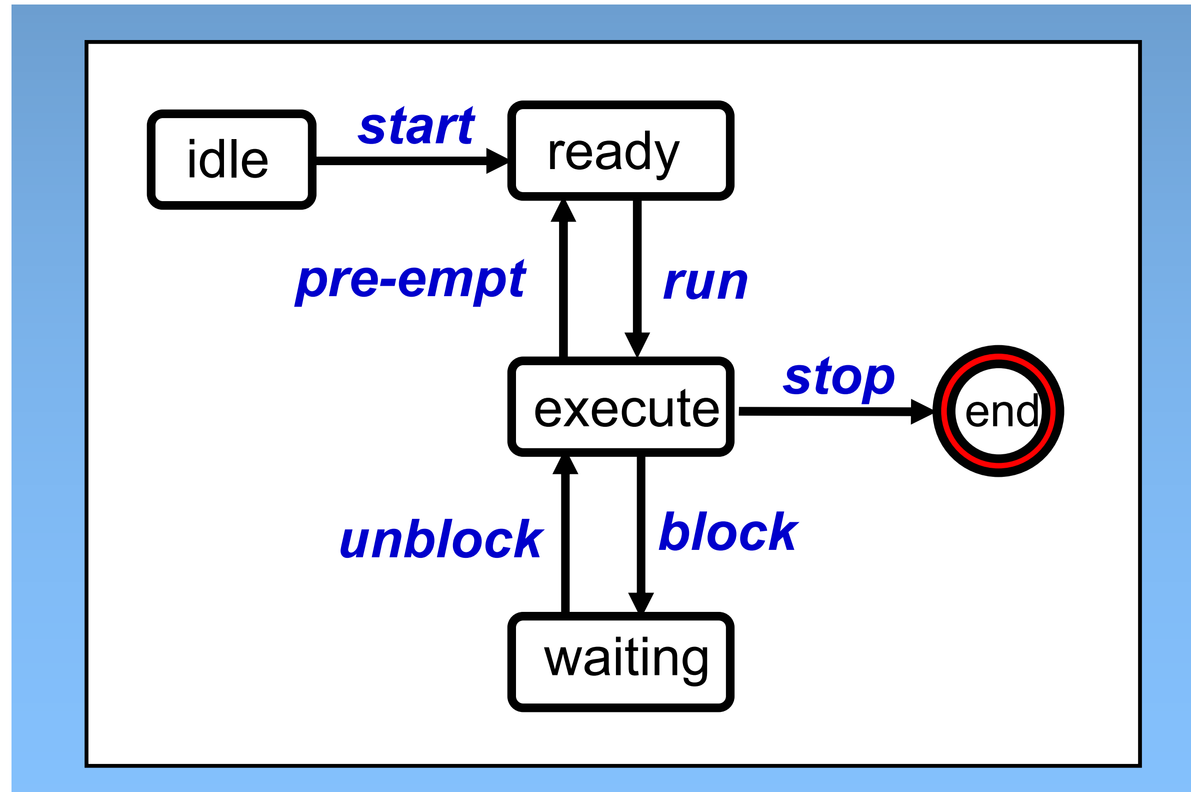
$$\sigma = \{(s_0, l_0, s_1), (s_1, l_1, s_2), \ldots\}$$

such that

$$\forall i, i \geq 0 \; s.th. \; (s_i, l_i, s_{i+1}) \in T$$

Each run corresponds to a state sequence in S and a word in L

Definition of a Run
Example



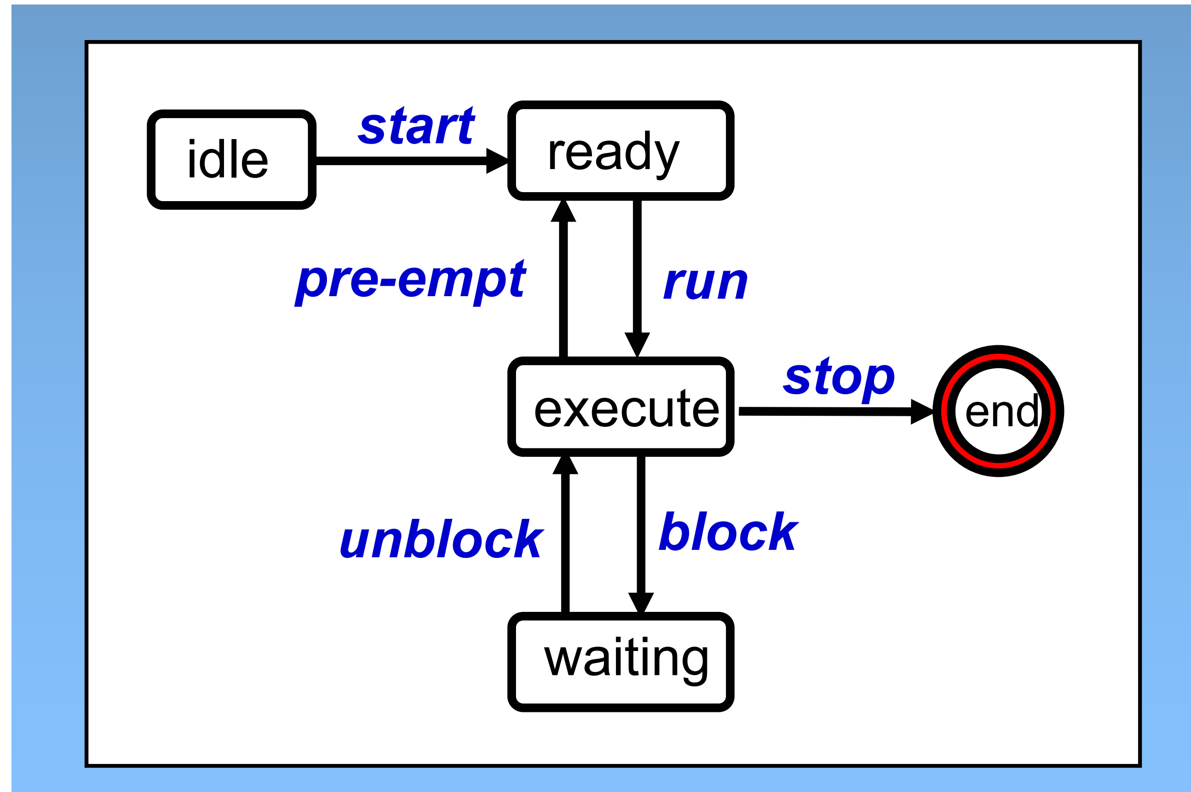
- A state sequence from a run: $\{idle, ready, \{execute, waiting\}^*\}$
- The corresponding word in L: $\{start, run, \{block, unblock\}^*\}$

Remarks:

- A single state sequence may correspond to more than one word

Definition of a Run

Example



- A state sequence from a run: $\{idle, ready, \{execute, waiting\}^*\}$
- The corresponding word in L: $\{start, run, \{block, unblock\}^*\}$

Remarks:

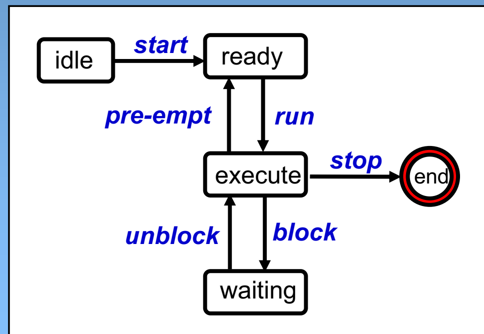
- A single state sequence may correspond to more than one word

Definition of Acceptance

Definition

An accepting run of a finite state automaton $A = (S, s_0, L, F, T)$ is a finite run σ in which the final transition (s_{n-1}, l_{n-1}, s_n) has the property that $s_n \in A.F$

Example:



state sequence of an *accepting* run:
{ idle, ready, execute, waiting, execute, end }

the corresponding *word* in L:
{*start, run, block, unblock, stop* }

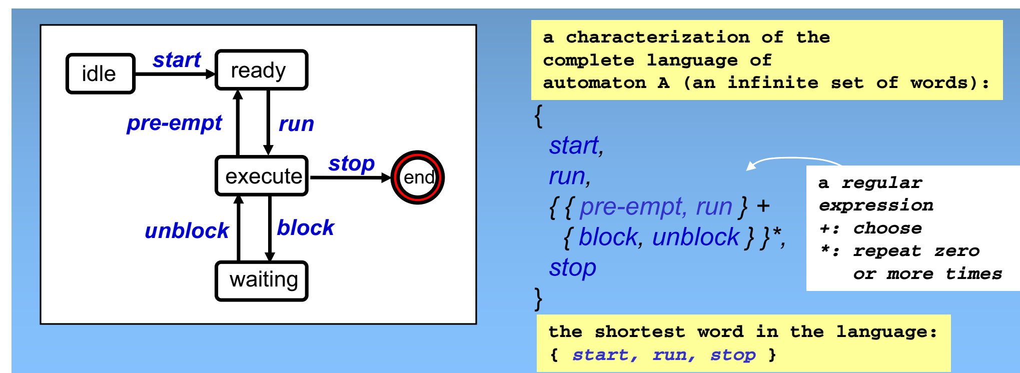
Language Accepted by an Automaton

Definition

The language $\mathcal{L}(A)$ of automaton $A = (S, s_0, L, F, T)$ is the set of words in $A.L$ that correspond to the set of all the accepting runs of A

Notice that there can be infinitely many words in the language of even a small finite state automaton

Example:

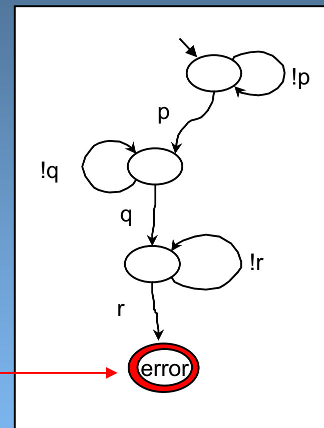


Reasoning about Runs

sample property:

"if first *p* becomes *true*
and then later *q* becomes *true*,
then *r* can no longer become *true*"

interpretation:

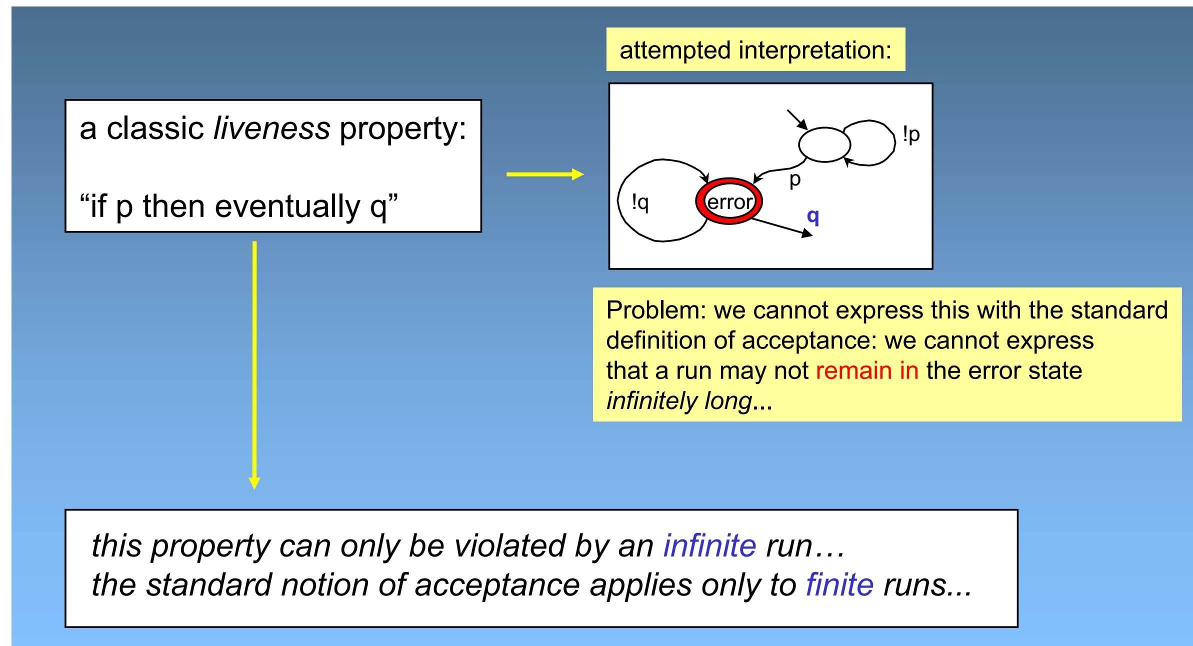


reaching this state constitutes a complete match of the pattern that specifies the correctness violation

correctness claim:
it is an error if in a run we see first p then q and then r

this property is easily expressed with the standard definition of acceptance

Reasoning about *Infinite* Runs



We need, thus, to extend the notion of run, acceptance, ...

Büchi Acceptance

- An infinite run is often called an ω-run ("omega run")
- An acceptance property for ω-runs are called ω-acceptance and can be defined in different ways
  - The so-called Büchi, Müller, Rabin, Streett, etc, acceptance conditions
  - We adopt here the one introduced by Büchi [?] [?]

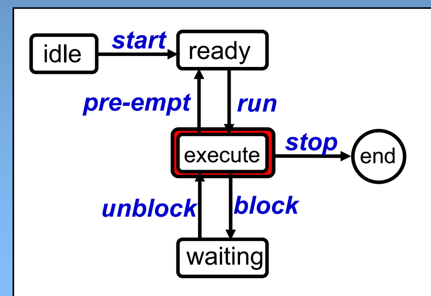
Definition

An accepting ω-run of finite state automaton $A = (S, s_0, L, F, T)$ is an infinite run σ such that

$$\exists i \geq 0, (s_{i-1}, l_{i-1}, s_i) \in \sigma \ s.th. \ s_i \in A.F \wedge s_i \in \sigma^\omega$$

i.e., at *least one state* in $A.F$ is visited infinitely often.
Automata with the above acceptance condition are called Büchi automata

Büchi Automata
Example



an accepting ω-run for this automaton:

{ idle, ready, {execute, ready}* }

the corresponding ω-word:

{*start*, *run*, { *pre-empt*, *run*}* }

the ω-language of an automaton is the set of all ω-words accepted

Generalized Büchi Automata

Definition

A generalized Büchi automaton is an automaton $A = (S, s_0, L, F, T)$, where $F \subseteq 2^S$ ($F = \{f_1, \ldots, f_n\}$ and $f_i \subseteq S$). A run σ of A is accepting if

$$\text{for each } f_i \in F, inf(\sigma) \cap f_i \neq \emptyset.$$

- A generalized Büchi Automaton differs from a Büchi Automaton by allowing multiple accepting sets instead of only one
- Generalized Büchi automata are not more expressive than usual Büchi automata

Generalized Büchi Automata

Definition

A generalized Büchi automaton is an automaton $A = (S, s_0, L, F, T)$, where $F \subseteq 2^S$ ($F = \{f_1, \ldots, f_n\}$ and $f_i \subseteq S$).
A run σ of A is accepting if

$$\text{for each } f_i \in F, inf(\sigma) \cap f_i \neq \emptyset.$$

- A generalized Büchi Automaton differs from a Büchi Automaton by allowing multiple accepting sets instead of only one
- Generalized Büchi automata are not more expressive than usual Büchi automata

The Stutter Extension Rule

- It would be convenient to include the acceptance for finite runs as a special case of acceptance for infinite runs - For that we need:
  - Let ε be a predefined nil symbol
  - The label set of the automaton is extended to $L \cup \{\varepsilon\}$
  - To determine ω-acceptance, a finite run is (thought to be) extended into an equivalent infinite run by stuttering the final state on ε

Definition

The stutter extension of a finite run σ with final state s_n, is the ω-run

$$\sigma \ (s_n, \varepsilon, s_n)^{\omega}$$

The Stutter Extension Rule

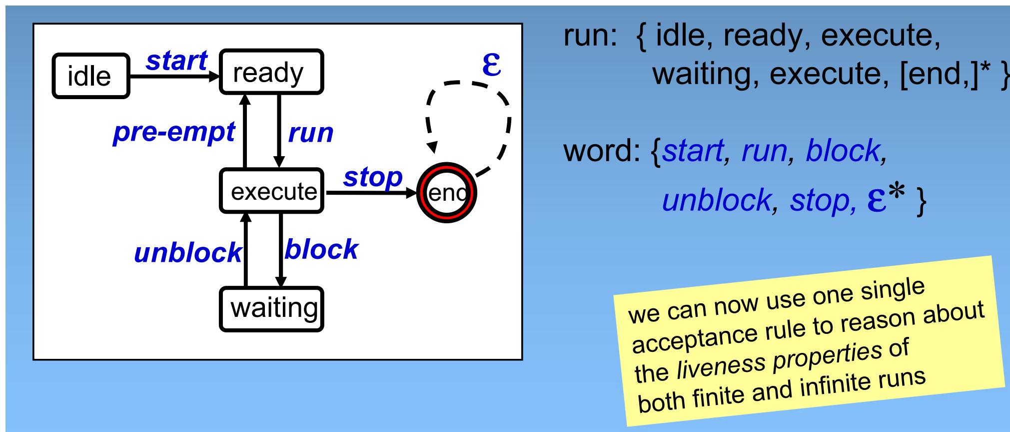
- It would be convenient to include the acceptance for finite runs as a special case of acceptance for infinite runs - For that we need:
  - Let ε be a predefined nil symbol
  - The label set of the automaton is extended to $L \cup \{\varepsilon\}$
  - To determine ω-acceptance, a finite run is (thought to be) extended into an equivalent infinite run by stuttering the final state on ε

Definition

The stutter extension of a finite run σ with final state s_n, is the ω-run

$$\sigma \; (s_n, \varepsilon, s_n)^\omega$$

The Stutter Extension Rule
Example



Satisfaction (semantics)

Definition

We define the notion that an LTL formula φ is true (false) relative to a path σ, written $\sigma \models \varphi$ ($\sigma \not\models \varphi$) as follows.

$$\begin{array}{lll} \sigma \models \varphi & \text{iff} & \sigma_0 \Vdash \varphi \text{ when } \varphi \in \mathcal{L} \\ \sigma \models \neg\varphi & \text{iff} & \sigma \not\models \varphi \\ \sigma \models \varphi \vee \psi & \text{iff} & \sigma \models \varphi \text{ or } \sigma \models \psi \\ & & \\ \sigma \models \Box\varphi & \text{iff} & \sigma^k \models \varphi \text{ for all } k \geq 0 \\ \sigma \models \Diamond\varphi & \text{iff} & \sigma^k \models \varphi \text{ for some } k \geq 0 \\ \sigma \models \bigcirc\varphi & \text{iff} & \sigma^1 \models \varphi \end{array}$$

(cont.)

Satisfaction (semantics) (2)

Definition

(cont.)

$$\sigma \models \varphi U \psi \quad \text{iff} \quad \sigma^k \models \psi \text{ for some } k \geq 0 \text{, and}$$
$$\sigma^i \models \varphi \text{ for every } i \text{ such that } 0 \leq i < k$$

$$\sigma \models \varphi R \psi \quad \text{iff} \quad \text{for every } j \geq 0,$$
$$\text{if } \sigma^i \not\models \varphi \text{ for every } i < j \text{ then } \sigma^j \models \psi$$

$$\sigma \models \varphi W \psi \quad \text{iff} \quad \sigma \models \varphi U \psi \text{ or } \sigma \models \Box \varphi$$

From Kripke Structures to Büchi Automata

- LTL formulas can be interpreted on sets of infinite runs of Kripke structures

We recall the definition (slightly different from previous lecture)

Definition

A Kripke structure M is a four-tuple (W, R, W_0, V) where

- W is a finite non-empty set of states (*worlds*)
- $R \subseteq W \times W$ is a total accessibility relation between states (transition relation)
- $W_0 \subseteq W$ is the set of starting states
- $V : W \longrightarrow 2^{AP}$ is a map labeling each state with a set of propositional variables

A path in M is an infinite sequence $\sigma = w_0, w_1, w_2, \ldots$ of worlds such that for every $i \geq 0$, $w_i R w_{i+1}$. One can think of a path as an infinite branch in a tree corresponding to the unwind of the Kripke structure.

From Kripke Structures to Büchi Automata

Obtaining the automaton

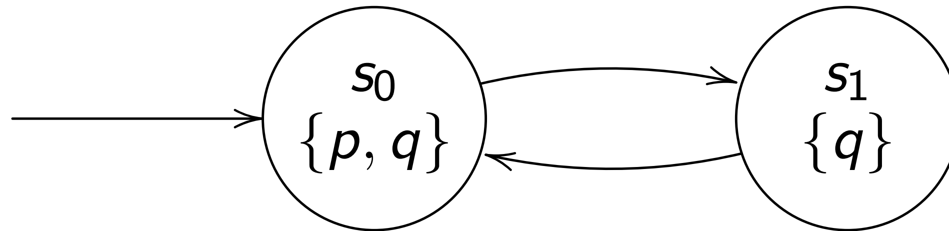
- An ω-regular automaton $A = (S, s_0, L, F, T)$ can be obtained from a Kripke structure $M = (W, R, W_0, V)$ as follows

- $S = W \cup \{i\}$
- $s_0 = \{i\}$
- $L = 2^{AP}$
- $F = W \cup \{i\}$
- For $s, s' \in S$ $s.th.$ $(s, l, s') \in T$ iff $(s, s') \in R \wedge l = V(s')$
- $(i, l, s) \in T$ iff $s \in W_0 \wedge l = V(s)$

From Kripke Structures to Büchi Automata

Example

A Kripke structure (whose only infinite run is a model to $\Box q$ and $\Box\Diamond p$, for instance):

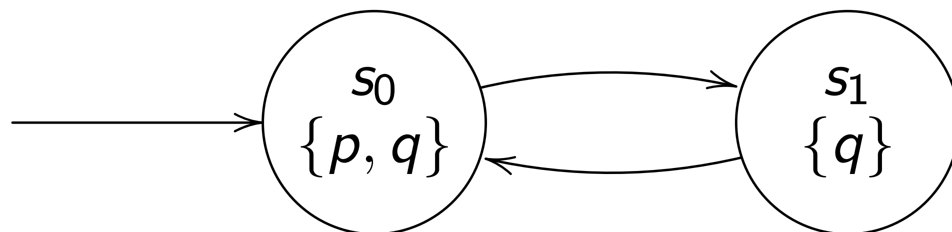


The corresponding Büchi Automaton:

From Kripke Structures to Büchi Automata

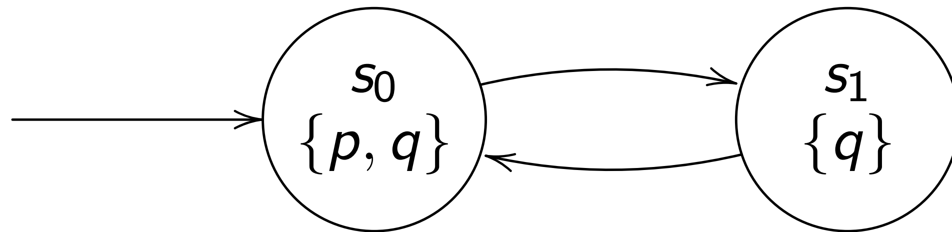
Example

A Kripke structure (whose only infinite run is a model to $\Box q$ and $\Box\Diamond p$, for instance):



The corresponding Büchi Automaton:

A Kripke structure (whose only infinite run is a model to $\Box q$ and $\Box\Diamond p$, for instance):



The corresponding Büchi Automaton:

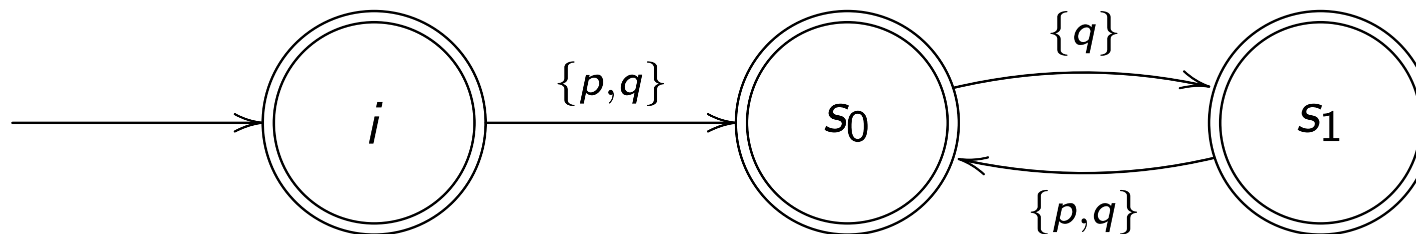


Figure: Büchi-Automaton

From Logic to Automata

- For any LTL formula ψ there exists a Büchi automaton that accepts precisely those runs for which the formula ψ is satisfied
- **Example:** The formula $\Diamond\Box p$ corresponds to the following nondeterministic Büchi automaton:

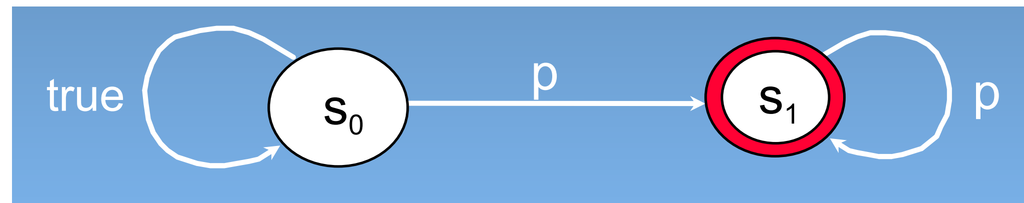


Figure: From LTL to automata

We will see the algorithm next lecture... For the moment, believe me that it is indeed the case

From Logic to Automata

- For any LTL formula ψ there exists a Büchi automaton that accepts precisely those runs for which the formula ψ is satisfied
- **Example:** The formula $\Diamond\Box p$ corresponds to the following nondeterministic Büchi automaton:

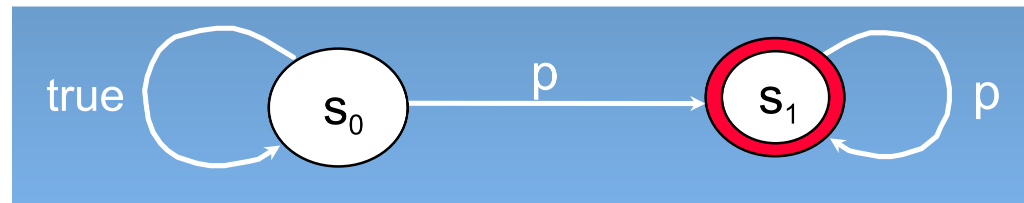
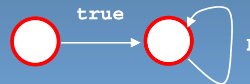


Figure: From LTL to automata

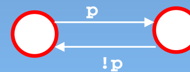
We will see the algorithm next lecture... For the moment, believe me that it is indeed the case

Omega-regular Properties

- something not expressible in pure LTL:
  - (p) *can* hold after an even number of execution steps, but *never* holds after an odd number of steps
  - [] X (p) certainly does not capture it:



  - p && [](p -> X!p) && [](!p -> Xp) does not capture it either (because now p *must* always hold after all even steps):



(ltl2ba -f)

  - ∃t, t && [] (t -> X !t) && [] (!t -> Xt) && [] (!t -> !p)
    this formula expresses it correctly

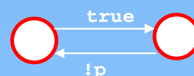


Figure: ω-regular properties

Expressiveness

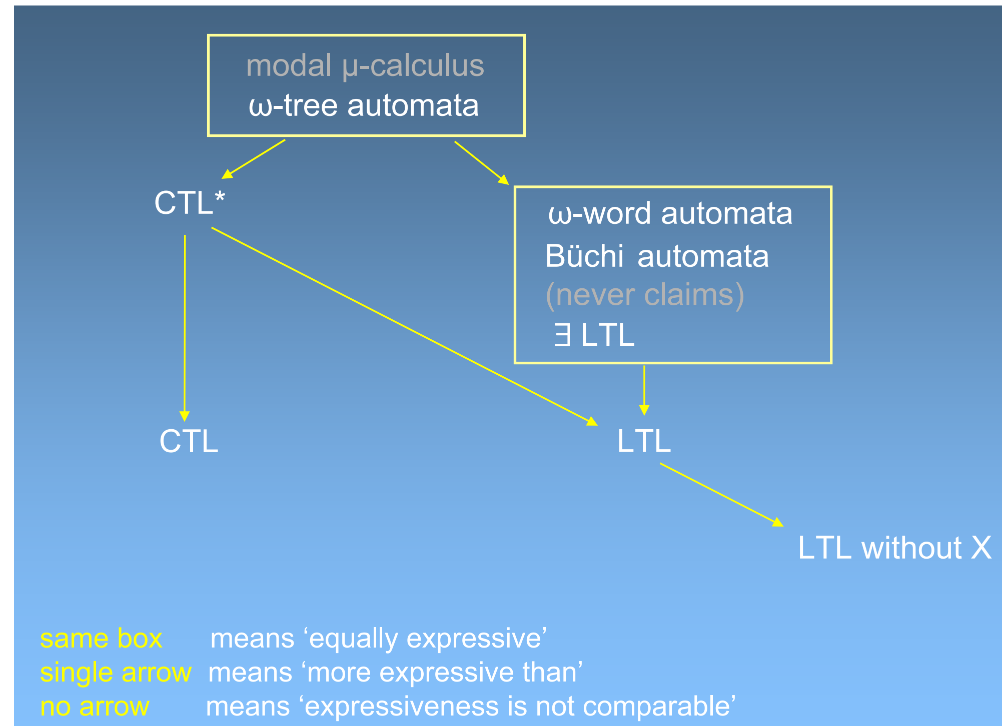


Figure: Expressiveness

Implications in Model Checking

- At the beginning we said that the automata-based model checking method was based on the following check:

$$\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset$$

where S is a model of the system and P of the property

- So, the following Büchi automata's decidable properties are important for model checking
  - Language emptiness: are there any accepting runs?
  - Language intersection: are there any runs accepted by two or more automata?
  - Language complementation

How does it work?

Implications in Model Checking

- At the beginning we said that the automata-based model checking method was based on the following check:

$$\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset$$

  where S is a model of the system and P of the property
- So, the following Büchi automata's decidable properties are important for model checking
  - Language emptiness: are there any accepting runs?
  - Language intersection: are there any runs accepted by two or more automata?
  - Language complementation

How does it work?

Implications in Model Checking

- At the beginning we said that the automata-based model checking method was based on the following check:

$$\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset$$

  where S is a model of the system and P of the property
- So, the following Büchi automata's decidable properties are important for model checking
  - Language emptiness: are there any accepting runs?
  - Language intersection: are there any runs accepted by two or more automata?
  - Language complementation

How does it work?

Implications for Model Checking

In theory:

- The system is represented as a Büchi automaton A
  - The automaton corresponds to the asynchronous product of automata $A_1, \ldots, A_n$ (representing the asynchronous processes)

$$A = \prod_{i=1}^{n} A_i$$

- The property is originally given as an LTL formula ψ
- The property ψ is translated into a Büchi automaton B[4]
- We perform the following check:

$$\mathcal{L}(A) \cap \overline{\mathcal{L}(B)} = \emptyset$$

But... complementing a Büchi automaton is difficult!

[4] Alternatively, the property can be given directly as a Büchi automaton

Implications for Model Checking

In theory:

- The system is represented as a Büchi automaton A
  - The automaton corresponds to the asynchronous product of automata $A_1, \ldots, A_n$ (representing the asynchronous processes)

$$A = \prod_{i=1}^{n} A_i$$

- The property is originally given as an LTL formula ψ
- The property ψ is translated into a Büchi automaton B[4]
- We perform the following check:

$$\mathcal{L}(A) \cap \overline{\mathcal{L}(B)} = \emptyset$$

But... complementing a Büchi automaton is difficult!

[4] Alternatively, the property can be given directly as a Büchi automaton

Implications in Model Checking

In practice (e.g., in SPIN) we want to avoid automata complementation:

- Assume A as before
- The negation of the property ψ is automatically translated into a Büchi automaton $\overline{B}$ (since $\overline{\mathcal{L}(B)} \equiv \mathcal{L}(\overline{B})$)
- By making the synchronous product of A and $\overline{B}$ ($\overline{B} \otimes A$) we can check whether the system satisfies the property

$$\mathcal{L}(A) \cap \mathcal{L}(\overline{B}) = \emptyset$$

- If the intersection is empty, the property ψ holds for A
- Otherwise, use an accepted word of the nonempty intersection as a counterexample

Asynchronous Product

Definition

The asynchronous product $\prod$ of a finite set of finite automata $A_1, \ldots A_n$ is a new finite state automaton $A = (S, s_0, L, T, F)$ where:

- $A.S$ is the Cartesian product $A_1.S \times A_2.S \times \ldots \times A_n.S$
- $A.s_0$ is the n-tuple $(A_1.s_0, A_2.s_0, \ldots, A_n.s_0)$
- $A.L$ is the union set $A_1.L \cup A_2.L \cup \ldots \cup A_n.L$
- $A.T$ is the set of tuples $((x_1, \ldots, x_n), l, (y_1, \ldots, y_n))$ such that $\exists i, 1 \leq i \leq n, (x_i, l, y_i) \in A_i.T$ and $\forall j, 1 \leq j \leq n, j \neq i \implies (x_j \equiv y_j)$
- $A.F$ contains those states from $A.S$ that satisfy $\forall (A_1.s, A_2.s, \ldots, A_n.s) \in A.F, \exists i, 1 \leq i \leq n, A_i.s \in A_i.F$

Asynchronous Product
Example

- Assume two non-terminating asynchronous processes A_1 and A_2:
  - A_1 tests whether the value of a variable x is odd, in which case updates it to $3 * x + 1$
  - A_2 tests whether the value of a variable x is even, in which case updates it to $x/2$
- Let ψ the following property: $\Box\Diamond(x \geq 4)$
  - The negation of the formula is:

Question: Given an initial value for x, does the property hold?

- Assume two non-terminating asynchronous processes A_1 and A_2:
  - A_1 tests whether the value of a variable x is odd, in which case updates it to $3 * x + 1$
  - A_2 tests whether the value of a variable x is even, in which case updates it to $x/2$
- Let ψ the following property: $\Box\Diamond(x \geq 4)$
  - The negation of the formula is:

Question: Given an initial value for x, does the property hold?

Asynchronous Product
Example

- Assume two non-terminating asynchronous processes A_1 and A_2:
  - A_1 tests whether the value of a variable x is odd, in which case updates it to $3 * x + 1$
  - A_2 tests whether the value of a variable x is even, in which case updates it to $x/2$
- Let ψ the following property: $\Box\Diamond(x \geq 4)$
  - The negation of the formula is: $\Diamond\Box(x < 4)$

Question: Given an initial value for x, does the property hold?

- Assume two non-terminating asynchronous processes A_1 and A_2:
  - A_1 tests whether the value of a variable x is odd, in which case updates it to $3 * x + 1$
  - A_2 tests whether the value of a variable x is even, in which case updates it to $x/2$
- Let ψ the following property: $\Box\Diamond(x \geq 4)$
  - The negation of the formula is: $\Diamond\Box(x < 4)$

Question: Given an initial value for x, does the property hold?

Remark

In Promela semantics an expression statement has to evaluate to non-zero to be executable. So to test whether a variable `x` is odd we write `!(x%2)`, and `(x%2)` for checking whether `x` is even.

Given `x=4`, `!(4%2)` evaluates to `!(0)` or written more clearly as `!(false)` which is `(true)`.

Remark

In Promela semantics an expression statement has to evaluate to non-zero to be executable. So to test whether a variable `x` is odd we write `!(x%2)`, and `(x%2)` for checking whether `x` is even. Given `x=4`, `!(4%2)` evaluates to `!(0)` or written more clearly as `!(false)` which is `(true)`.

Asynchronous Product
Example

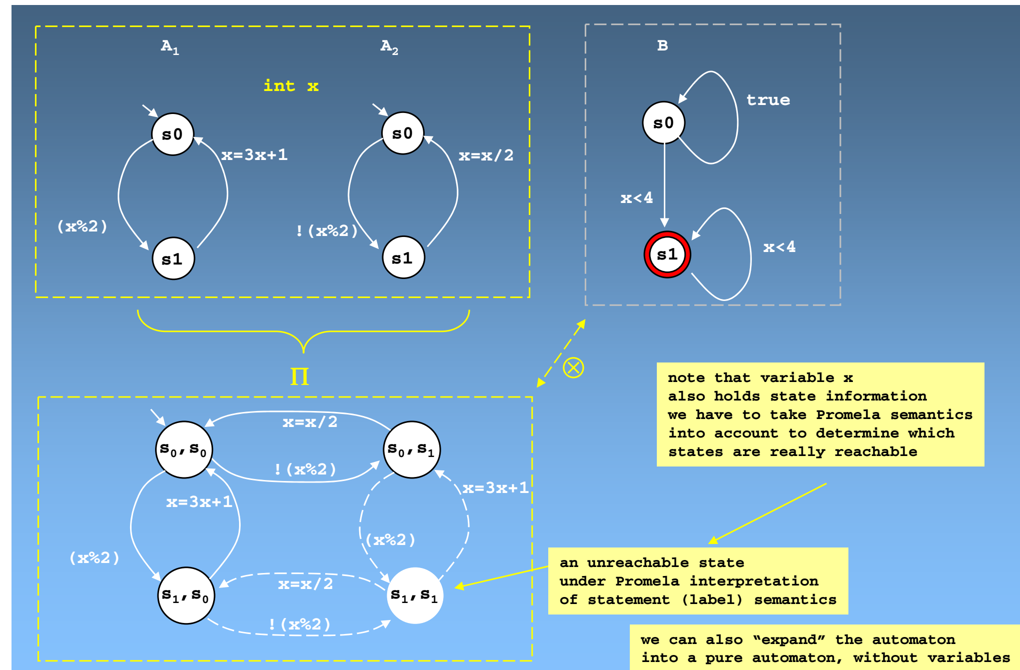


Figure: Asynchronous product

Asynchronous Product
Example

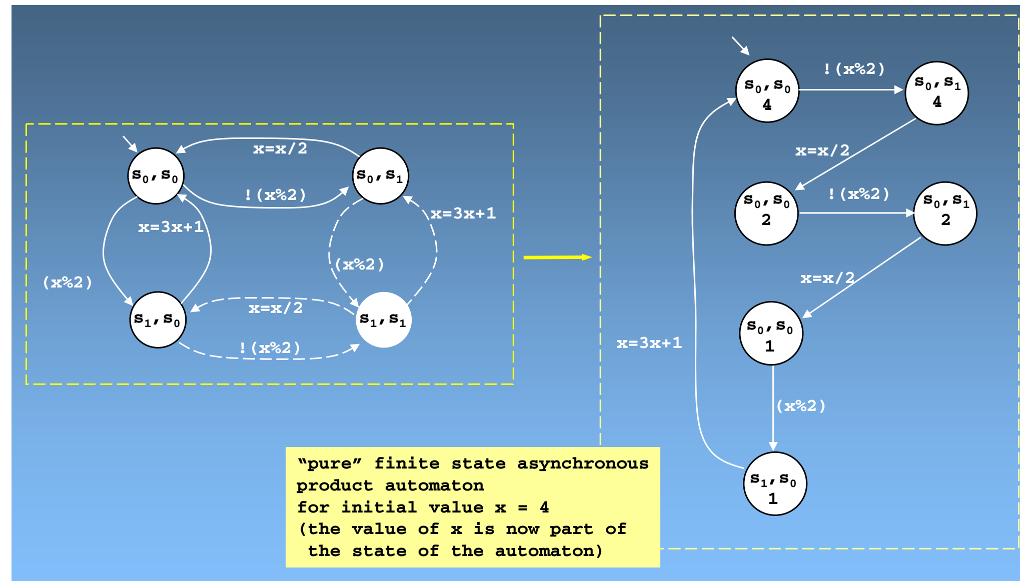


Figure: Asynchronous product

Remarks

- Not all the states in $A.S$ are necessary reachable from $A.s_0$
  - Their reachability depends on the semantics given to the labels in $A.L$ (the interpretation of the labels depends on Promela semantics as we'll see in a future lecture)
- The transitions in the product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time
  - This gives an interleaving semantics of the processes
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)
  - Some transitions may synchronize by sending and receiving a message
- For hardware verification, the asynchronous product is defined differently: **each** of the components with enabled transitions is making a transition (simultaneously)

Asynchronous Product

Remarks

- Not all the states in $A.S$ are necessary reachable from $A.s_0$
  - Their reachability depends on the semantics given to the labels in $A.L$ (the interpretation of the labels depends on Promela semantics as we'll see in a future lecture)
- The transitions in the product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time
  - This gives an interleaving semantics of the processes
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)
  - Some transitions may synchronize by sending and receiving a message
- For hardware verification, the asynchronous product is defined differently: **each** of the components with enabled transitions is making a transition (simultaneously)

Asynchronous Product

Remarks

- Not all the states in $A.S$ are necessary reachable from $A.s_0$
  - Their reachability depends on the semantics given to the labels in $A.L$ (the interpretation of the labels depends on Promela semantics as we'll see in a future lecture)
- The transitions in the product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time
  - This gives an interleaving semantics of the processes
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)
  - Some transitions may synchronize by sending and receiving a message
- For hardware verification, the asynchronous product is defined differently: **each** of the components with enabled transitions is making a transition (simultaneously)

Asynchronous Product

Remarks

- Not all the states in $A.S$ are necessary reachable from $A.s_0$
  - Their reachability depends on the semantics given to the labels in $A.L$ (the interpretation of the labels depends on Promela semantics as we'll see in a future lecture)
- The transitions in the product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time
  - This gives an interleaving semantics of the processes
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)
  - Some transitions may synchronize by sending and receiving a message
- For hardware verification, the asynchronous product is defined differently: **each** of the components with enabled transitions is making a transition (simultaneously)

Asynchronous Product

Remarks

- Not all the states in $A.S$ are necessary reachable from $A.s_0$
  - Their reachability depends on the semantics given to the labels in $A.L$ (the interpretation of the labels depends on Promela semantics as we'll see in a future lecture)
- The transitions in the product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time
  - This gives an interleaving semantics of the processes
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)
  - Some transitions may synchronize by sending and receiving a message
- For hardware verification, the asynchronous product is defined differently: **each** of the components with enabled transitions is making a transition (simultaneously)

Asynchronous Product

Remarks

- Not all the states in $A.S$ are necessary reachable from $A.s_0$
  - Their reachability depends on the semantics given to the labels in $A.L$ (the interpretation of the labels depends on Promela semantics as we'll see in a future lecture)
- The transitions in the product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time
  - This gives an interleaving semantics of the processes
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)
  - Some transitions may synchronize by sending and receiving a message
- For hardware verification, the asynchronous product is defined differently: **each** of the components with enabled transitions is making a transition (simultaneously)

Asynchronous Product

Remarks

- Not all the states in $A.S$ are necessary reachable from $A.s_0$
  - Their reachability depends on the semantics given to the labels in $A.L$ (the interpretation of the labels depends on Promela semantics as we'll see in a future lecture)
- The transitions in the product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time
  - This gives an interleaving semantics of the processes
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)
  - Some transitions may synchronize by sending and receiving a message
- For hardware verification, the asynchronous product is defined differently: **each** of the components with enabled transitions is making a transition (simultaneously)

Synchronous Product

Definition

The synchronous product $\otimes$ of a finite set of two finite automata P and B is a new finite state automaton $A = (S, s_0, L, T, F)$ where:

- $A.S$ is the Cartesian product $P'.S \times B.S$ where P' is the *stutter closure* of P
  - A self-loop labeled with ε is attached to every state in P without outgoing transitions in $P.T$)
- $A.s_0$ is the pair $(P.s_0, B.s_0)$
- $A.L$ is the set of pairs (l_1, l_2) such that $l_1 \in P'.L$ and $l_2 \in B.L$
- $A.T$ is the set of pairs (t_1, t_2) such that $t_1 \in P'.T$ and $t_2 \in B.T$
- $A.F$ is the set of pairs (s_1, s_2) such that $s_1 \in P'.F$ or $s_2 \in B.F$

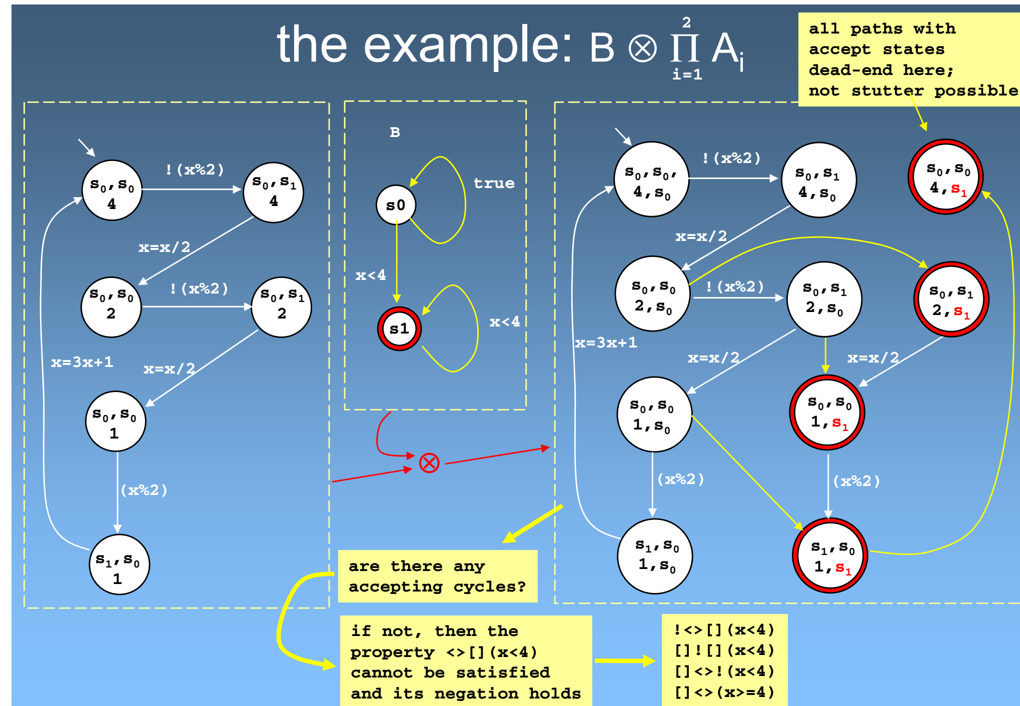


Figure: Synchronous product

Remarks

- We require the stutter-closure of P since P is a finite state automaton (the asynchronous product of the processes automata) and B is a standard Büchi automaton obtained form a LTL formula
- Not all the states in $A.S$ or $A.F$ are necessary reachable from $A.s_0$
- The main difference between asynchronous and synchronous products are on the definitions of L and T – In a synchronous product:
  - The transitions correspond to *joint* transitions of the component automata
  - The labels are pairs: the combination of the two labels of the original transitions in the component automata
- In general $P \otimes B \neq B \otimes P$, but given that in SPIN B is particular kind of automaton (labels are state properties, not actions), we have then $P \otimes B \equiv B \otimes P$

Synchronous Product

Remarks

- We require the stutter-closure of P since P is a finite state automaton (the asynchronous product of the processes automata) and B is a standard Büchi automaton obtained form a LTL formula
- Not all the states in $A.S$ or $A.F$ are necessary reachable from $A.s_0$
- The main difference between asynchronous and synchronous products are on the definitions of L and T – In a synchronous product:
  - The transitions correspond to *joint* transitions of the component automata
  - The labels are pairs: the combination of the two labels of the original transitions in the component automata
- In general $P \otimes B \neq B \otimes P$, but given that in SPIN B is particular kind of automaton (labels are state properties, not actions), we have then $P \otimes B \equiv B \otimes P$

Synchronous Product

Remarks

- We require the stutter-closure of P since P is a finite state automaton (the asynchronous product of the processes automata) and B is a standard Büchi automaton obtained form a LTL formula
- Not all the states in $A.S$ or $A.F$ are necessary reachable from $A.s_0$
- The main difference between asynchronous and synchronous products are on the definitions of L and T – In a synchronous product:
  - The transitions correspond to *joint* transitions of the component automata
  - The labels are pairs: the combination of the two labels of the original transitions in the component automata
- In general $P \otimes B \neq B \otimes P$, but given that in SPIN B is particular kind of automaton (labels are state properties, not actions), we have then $P \otimes B = B \otimes P$

Synchronous Product

Remarks

- We require the stutter-closure of P since P is a finite state automaton (the asynchronous product of the processes automata) and B is a standard Büchi automaton obtained form a LTL formula
- Not all the states in $A.S$ or $A.F$ are necessary reachable from $A.s_0$
- The main difference between asynchronous and synchronous products are on the definitions of L and T – In a synchronous product:
  - The transitions correspond to *joint* transitions of the component automata
  - The labels are pairs: the combination of the two labels of the original transitions in the component automata
- In general $P \otimes B \not\equiv B \otimes P$, but given that in SPIN B is particular kind of automaton (labels are state properties, not actions), we have then $P \otimes B \equiv B \otimes P$

Model Checking Algorithm

Strongly-connected components

Definition

A subset $S' \subseteq S$ in a directed graph is strongly-connected if there is a path between any pair of nodes in S', passing only through nodes in S'.
A strongly-connected component (SCC) is a maximal set of such nodes, i.e. it is not possible to add any node to that set and still maintain strong connectivity

Strongly-connected Components
Example

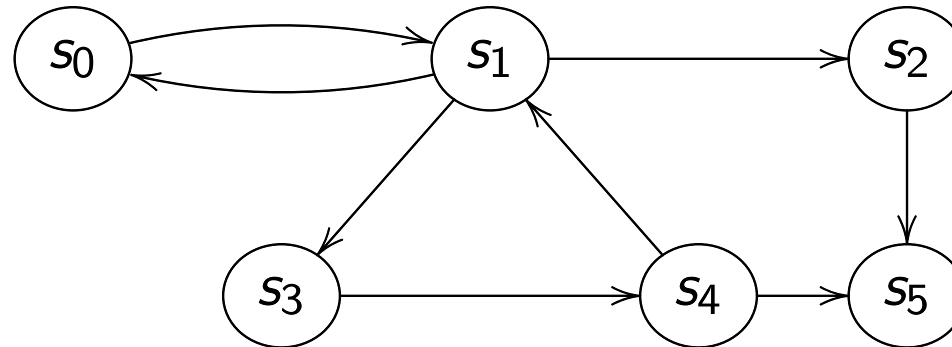


Figure: Strongly connected component

- Strongly-connected subsets:
- Strongly-connected components:

Strongly-connected Components

Example

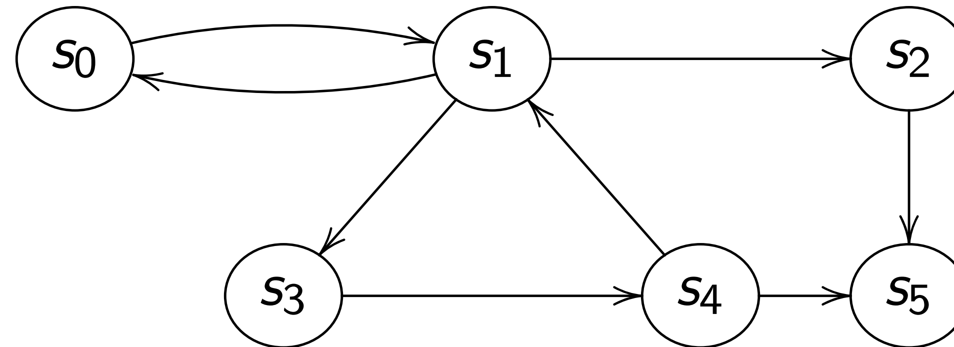


Figure: Strongly connected component

- Strongly-connected subsets:
- Strongly-connected components:

Strongly-connected Components
Example

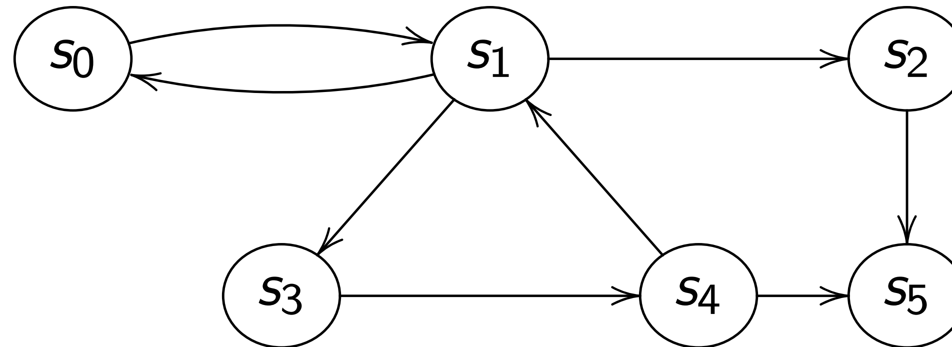


Figure: Strongly connected component

- Strongly-connected subsets:
  $S = \{s_0, s_1\}, \quad S' = \{s_1, s_3, s_4\}, \quad S'' = \{s_0, s_1, s_3, s_4\}$
- Strongly-connected components:

Strongly-connected Components

Example

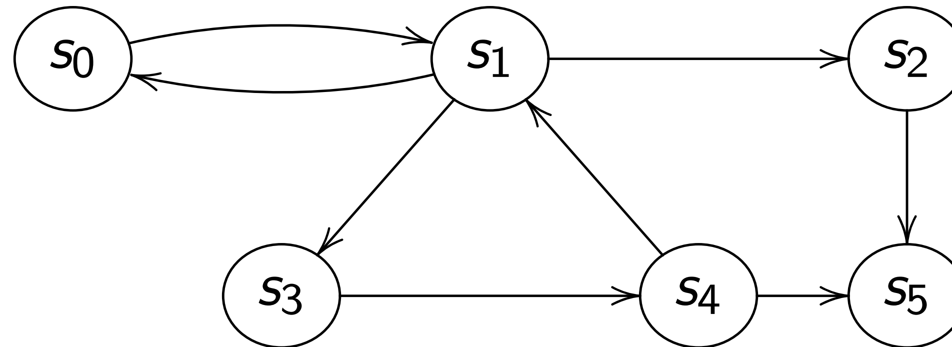


Figure: Strongly connected component

- Strongly-connected subsets:
  $S = \{s_0, s_1\}, \quad S' = \{s_1, s_3, s_4\}, \quad S'' = \{s_0, s_1, s_3, s_4\}$
- Strongly-connected components: Only $S'' = \{s_0, s_1, s_3, s_4\}$

Checking Emptiness

- Let σ be an accepting run of a Büchi automaton $A = (S, s_0, L, T, F)$
  - Since S is finite, there is some suffix σ' of σ s.t. every state on σ' is reachable from any other state on σ'
  - I.e., the states on σ' are contained in a SCC of the graph of A
  - This component is reachable from an initial state and contains an accepting state
- Thus, checking non-emptiness of $\mathcal{L}(A)$ is equivalent to finding a SCC in the graph of A that is reachable from an initial state and contains an accepting state
  - There are different algorithms for finding SCC. E.g.:
    - Tarjan's version of the *depth-first search* (DFS) algorithm
    - SPIN *nested depth-first search* algorithm
- If the language $\mathcal{L}(A)$ is non-empty, then there is a counterexample which can be represented in a finite way
  - It is *ultimately periodic*, i.e., it is of the form $\sigma_1\sigma_2^\omega$, where σ_1 and σ_2 are finite sequences

Checking Emptiness

- Let σ be an accepting run of a Büchi automaton $A = (S, s_0, L, T, F)$
  - Since S is finite, there is some suffix σ' of σ s.t. every state on σ' is reachable from any other state on σ'
  - I.e., the states on σ' are contained in a SCC of the graph of A
  - This component is reachable from an initial state and contains an accepting state
- Thus, checking non-emptiness of $\mathcal{L}(A)$ is equivalent to finding a SCC in the graph of A that is reachable from an initial state and contains an accepting state
  - There are different algorithms for finding SCC. E.g.:
    - Tarjan's version of the *depth-first search* (DFS) algorithm
    - SPIN *nested depth-first search* algorithm
- If the language $\mathcal{L}(A)$ is non-empty, then there is a counterexample which can be represented in a finite way
  - It is *ultimately periodic*, i.e., it is of the form $\sigma_1\sigma_2^\omega$, where σ_1 and σ_2 are finite sequences

Checking Emptiness

- Let σ be an accepting run of a Büchi automaton $A = (S, s_0, L, T, F)$
  - Since S is finite, there is some suffix σ' of σ s.t. every state on σ' is reachable from any other state on σ'
  - I.e., the states on σ' are contained in a SCC of the graph of A
  - This component is reachable from an initial state and contains an accepting state
- Thus, checking non-emptiness of $\mathcal{L}(A)$ is equivalent to finding a SCC in the graph of A that is reachable from an initial state and contains an accepting state
  - There are different algorithms for finding SCC. E.g.:
    - Tarjan's version of the *depth-first search* (DFS) algorithm
    - SPIN *nested depth-first search* algorithm
- If the language $\mathcal{L}(A)$ is non-empty, then there is a counterexample which can be represented in a finite way
  - It is *ultimately periodic*, i.e., it is of the form $\sigma_1\sigma_2^\omega$, where σ_1 and σ_2 are finite sequences

Model Checking Algorithm

- Let A be the automaton specifying the system and $\overline{B}$ the automaton corresponding to the negation of the property ψ

1. Construct the intersection automaton $C = A \cap \overline{B}$
2. Apply an algorithm to find SCCs reachable from the initial states of C
3. If none of the SCCs found contains an accepting state
   - The model A satisfies the property/specification ψ
4. Otherwise,
   4.1 Take one strongly-connected component SC of C
   4.2 Construct a path σ_1 from an initial state of C to some accepting state s of SC
   4.3 Construct a cycle from s and back to itself (such cycle exists since SC is a strongly-connected component)
   4.4 Let σ_2 be such cycle, excluding its first state s
   4.5 Announce that $\sigma_1 \sigma_2^\omega$ is a counterexample that is accepted by A, but it is not allowed by the property/specification ψ

Model Checking Algorithm

- Let A be the automaton specifying the system and $\overline{B}$ the automaton corresponding to the negation of the property ψ

1. Construct the intersection automaton $C = A \cap \overline{B}$
2. Apply an algorithm to find SCCs reachable from the initial states of C
3. If none of the SCCs found contains an accepting state
   - The model A satisfies the property/specification ψ
4. Otherwise,
   4.1 Take one strongly-connected component SC of C
   4.2 Construct a path σ_1 from an initial state of C to some accepting state s of SC
   4.3 Construct a cycle from s and back to itself (such cycle exists since SC is a strongly-connected component)
   4.4 Let σ_2 be such cycle, excluding its first state s
   4.5 Announce that $\sigma_1\sigma_2^\omega$ is a counterexample that is accepted by A, but it is not allowed by the property/specification ψ

Model Checking Algorithm

- Let A be the automaton specifying the system and $\overline{B}$ the automaton corresponding to the negation of the property ψ

1. Construct the intersection automaton $C = A \cap \overline{B}$
2. Apply an algorithm to find SCCs reachable from the initial states of C
3. If none of the SCCs found contains an accepting state
   - The model A satisfies the property/specification ψ
4. Otherwise,
   4.1 Take one strongly-connected component SC of C
   4.2 Construct a path σ_1 from an initial state of C to some accepting state s of SC
   4.3 Construct a cycle from s and back to itself (such cycle exists since SC is a strongly-connected component)
   4.4 Let σ_2 be such cycle, excluding its first state s
   4.5 Announce that $\sigma_1 \sigma_2^\omega$ is a counterexample that is accepted by A, but it is not allowed by the property/specification ψ

Model Checking Algorithm

- Let A be the automaton specifying the system and $\overline{B}$ the automaton corresponding to the negation of the property ψ

1. Construct the intersection automaton $C = A \cap \overline{B}$
2. Apply an algorithm to find SCCs reachable from the initial states of C
3. If none of the SCCs found contains an accepting state
   - The model A satisfies the property/specification ψ
4. Otherwise,
   - 4.1 Take one strongly-connected component SC of C
   - 4.2 Construct a path σ_1 from an initial state of C to some accepting state s of SC
   - 4.3 Construct a cycle from s and back to itself (such cycle exists since SC is a strongly-connected component)
   - 4.4 Let σ_2 be such cycle, excluding its first state s
   - 4.5 Announce that $\sigma_1 \sigma_2^\omega$ is a counterexample that is accepted by A, but it is not allowed by the property/specification ψ

Model Checking Algorithm

- Let A be the automaton specifying the system and $\overline{B}$ the automaton corresponding to the negation of the property ψ

1. Construct the intersection automaton $C = A \cap \overline{B}$
2. Apply an algorithm to find SCCs reachable from the initial states of C
3. If none of the SCCs found contains an accepting state
   - The model A satisfies the property/specification ψ
4. Otherwise,
   - 4.1 Take one strongly-connected component SC of C
   - 4.2 Construct a path σ_1 from an initial state of C to some accepting state s of SC
   - 4.3 Construct a cycle from s and back to itself (such cycle exists since SC is a strongly-connected component)
   - 4.4 Let σ_2 be such cycle, excluding its first state s
   - 4.5 Announce that $\sigma_1 \sigma_2^{\omega}$ is a counterexample that is accepted by A, but it is not allowed by the property/specification ψ

Final Remarks

Observation

- In Peled's book "Software Reliability Methods" [Peled, 2001] the definition of a Büchi automaton is very similar to our Kripke structure, with the addition of acceptance states
  - There is a labeling of the states associating to each state a set of subsets of propositions (instead of having the propositions as transition labels)
- We have chosen to define Büchi Automata in the way we did since this definition is compatible with the implementation of SPIN
  - It was taken from Holzmann's book "The SPIN Model Checker" [**?**]

Observation

- We have defined synchronous and asynchronous automata products with the aim of using SPIN (based on Holzmann's book)
  - The definition of asynchronous product is intended to capture the idea of (software) asynchronous processes running concurrently
  - The synchronous product is defined between an automaton specifying the concurrent asynchronous processes and an automaton obtained from an LTL formula (or obtained from a Promela `never` claim)
  - The purpose for adding the stutter closure (in the definition of the synchronous product) is to make it possible to verify both properties of finite and infinite sequences with the same algorithm
- I.e., you might find different definitions in the literature!
  - In particular, in Peled's book the automata product is defined differently, since the definition of Büchi automata is different

Observation

- We have defined synchronous and asynchronous automata products with the aim of using SPIN (based on Holzmann's book)
    - The definition of asynchronous product is intended to capture the idea of (software) asynchronous processes running concurrently
    - The synchronous product is defined between an automaton specifying the concurrent asynchronous processes and an automaton obtained from an LTL formula (or obtained from a Promela `never` claim)
    - The purpose for adding the stutter closure (in the definition of the synchronous product) is to make it possible to verify both properties of finite and infinite sequences with the same algorithm
- I.e., you might find different definitions in the literature!
    - In particular, in Peled's book the automata product is defined differently, since the definition of Büchi automata is different

Further Reading

- The first two parts of this lecture were mainly based on Chap. 6 of Holzmann's book "The SPIN Model Checker"
  - Automata products: Appendix A
- The 3rd part was taken from Peled's book

For next lecture (10./17.03.2017): Read Chap. 6 of Peled's book, mainly section 6.8 on translating LTL into Automata

- We will see how to apply the algorithm to an example

References I

[Manna and Pnueli, 1992] Manna, Z. and Pnueli, A. (1992).
The temporal logic of reactive and concurrent systems—Specification.
Springer Verlag, New York.

[Peled, 2001] Peled, D. (2001).
Software Reliability Methods.
Springer Verlag.