

# INF5140 – Specification and Verification of Parallel Systems

Spring 2018

Institutt for informatikk, Universitetet i Oslo

February 23, 2018



# INF5140 – Specification and Verification of Parallel Systems

## Lecture 5 - Introduction to Logical Model Checking and Theoretical Foundations

Spring 2018

Institutt for informatikk, Universitetet i Oslo

February 23, 2018



## 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>)

## 1. Logic Model Checking: What is it about?

- The Basic Method

- General remarks

- Motivating Examples

## 2. Automata and Logic

- Finite State Automata

- Büchi Automata

- Something on Logic and Automata

- Implications in Model Checking

- Automata products

## 3. Model Checking Algorithm

- Preliminaries

- The Algorithm

## 4. Final Remarks

- Something on Automata

- LTL to Büchi

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 system 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<sup>1</sup>
    - The number of and size of buffered channels
    - The number of asynchronous processes

---

<sup>1</sup>and in other model checkers.

# The Basic Method

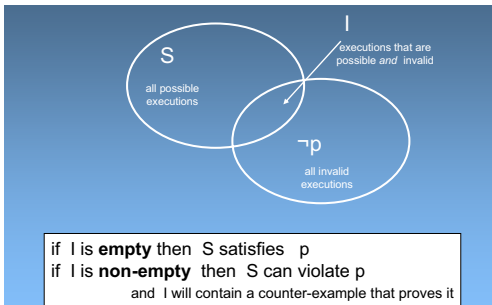
There are different model checking techniques. We will use the *automata-theoretic approach* which is implemented in the **SPIN** model checker (tool). Theoretically:

- 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



Graphically:

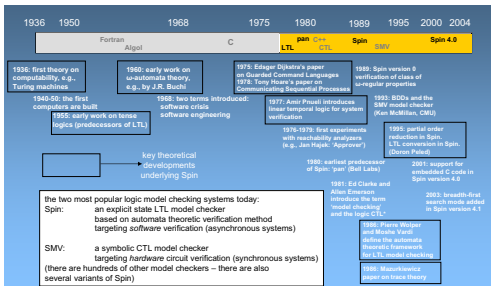


# Scope of the method

- 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 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)

# Distributed Algorithms

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?



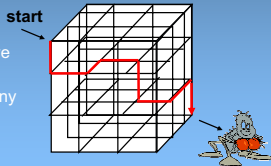
# Thread Interleaving

- the number of possible thread interleavings is...

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

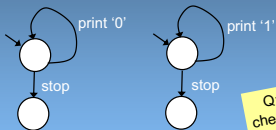
placing 3 sets of 3 tokens in 9 slots

- 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

## 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  $S$

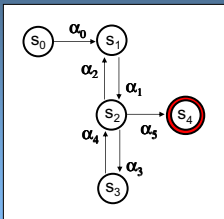
We will, in general, follow Holzmann's notation:  $A.S$  denotes the state  $S$  of automaton  $A$ ,  $A.T$  denotes the transition relation  $T$  of  $A$ , and so on....

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 = {  $\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5$  }

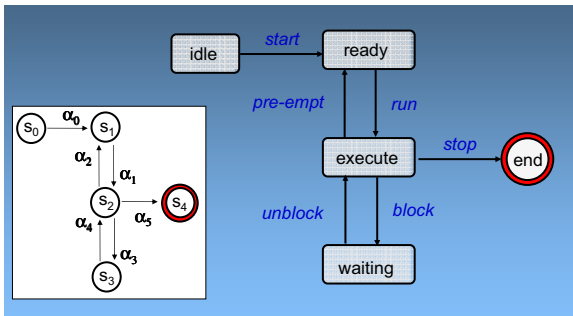
A.F = {  $s_4$  }

A.T = {  $(s_0, \alpha_0, s_1), (s_1, \alpha_1, s_2), \dots$  }

# 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*)

## 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  $T$

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

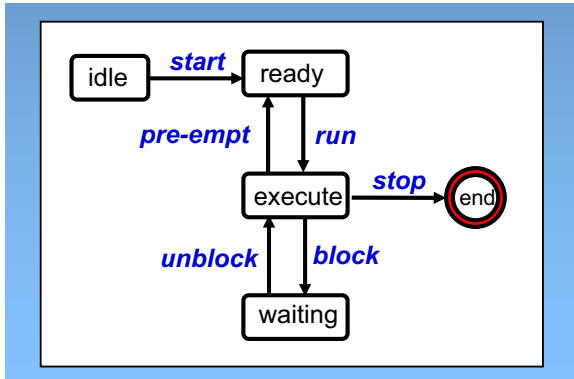
such that

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

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



## Example run



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

### Remarks:

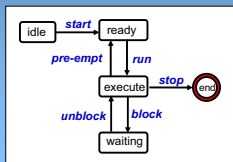
- A single state sequence may correspond to more than one word
- For non-deterministic automata, the same word may

# Definition of Acceptance

## Definition (Büchi acceptance)

An **accepting** run of a finite state automaton  $A = (S, s_0, L, F, T)$  is a finite run  $\sigma$  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* }

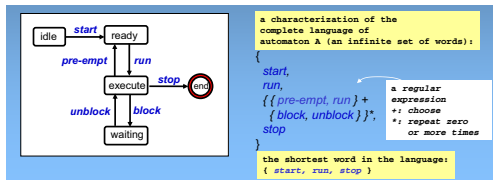
# Language accepted by an automaton

## 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  $A$

Note: there generally are infinitely many words in the language

**Example:**



# Reasoning about Runs

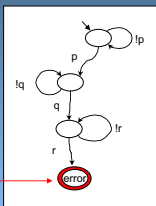
sample property:

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

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

this property is easily expressed with  
the standard definition of acceptance

interpretation:

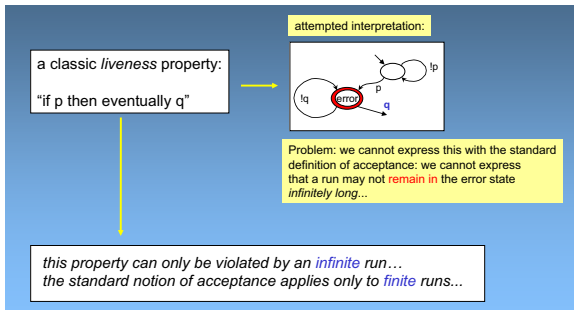


correctness claim:  
it is an error if in a run we  
see first  $p$  then  $q$  and then  $r$

## Comparison to FSA in “standard” language theory

- remember classical FSA (and regular expressions)
- (typically infinite) languages of finite words
- remember: accepting runs are finite
- in “classical” language theory: *infinite* words completely out of the picture

# Reasoning about *Infinite* Runs



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

# Büchi Acceptance

- infinite run: often called  $\omega$ -run (“omega run”)
- corresponding acceptance properties:  $\omega$ -acceptance
- different versions known
  - The so-called Büchi, Müller, Rabin, Streett, etc., acceptance conditions
  - Here for now: Büchi acceptance condition [?] [?]

## Definition (Büchi acceptance)

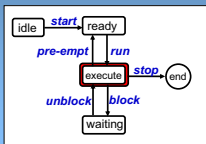
An **accepting  $\omega$ -run** of finite state automaton  $A = (S, s_0, L, F, T)$  is an infinite run  $\sigma$  such that

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

*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  $\omega$ -run for this automaton:  
{ idle, ready, {execute, ready}<sup>\*</sup> }

the corresponding  $\omega$ -word:  
{start, run, {pre-empt, run}<sup>\*</sup> }

the  $\omega$ -language of an automaton is  
the set of all  $\omega$ -words accepted



## Definition (Generalized Büchi automaton)

A **generalized Büchi automaton** is an automaton

$A = (S, s_0, L, F, T)$ , where  $F \subseteq 2^S$ .

Let  $F = \{f_1, \dots, f_n\}$  and  $f_i \subseteq S$ . A run  $\sigma$  of  $A$  is **accepting** if

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

- $\text{inf}(\sigma)$ : states visited infinitely often in
- generalized Büchi automaton: multiple accepting sets instead of only one ( $\neq$  “original” Büchi Automata)
- generalized Büchi automata: equally expressive

- treat finite and infinite acceptance uniformly.
- finite runs as infinite ones, where, at some point, infinitely often “nothing” happens (**stuttering**)
  - Let  $\varepsilon$  be a predefined nil symbol
  - alphabet/label set extended to  $L + \{\varepsilon\}$
  - extend finite run to an equivalent infinite run: keep on stuttering after the end of run<sup>2</sup>

## Definition (Stutter extension)

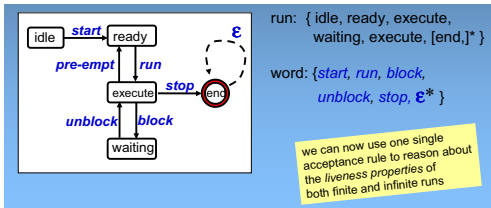
The **stutter extension** of a finite run  $\sigma$  with final state  $s_n$ , is the  $\omega$ -run

$$\sigma (s_n, \varepsilon, s_n)^\omega \quad (1)$$

---

<sup>2</sup>The run must end in a final state.

# Stuttering example



- LTL formulas can be interpreted on sets of infinite runs of Kripke structures
- Kripke structure/model:
  - “automaton” or “transition system”
  - transitions unlabelled (typically)
  - states (or worlds): “labelled”, in the most basic situation: sets of propositional variables

One can think of a path as an infinite branch in a tree corresponding to the unwind of the Kripke structure.

## Definition (Kripke structure)

A **Kripke structure**  $M$  is a four-tuple  $(S, R, S_0, V)$  where

- $S$  is a finite non-empty set of states (also “worlds”)
- $R \subseteq S \times S$  is a total relation between states (transition relation, aka accessibility relation)
- $S_0 \subseteq S$  is the set of starting states
- $V : S \rightarrow 2^{AP}$  is a map labeling each state with a set of propositional variables

Notation:  $\longrightarrow$  for accessibility relation

A **path** in  $M$  is an infinite sequence  $\sigma = s_0, s_1, s_2, \dots$  of states such that for every  $i \geq 0$ ,  $s_i \longrightarrow s_{i+1}$ .

- “subtle” differences
- labelled transitions vs. labelled states
- easy to transform one representation into the other
- here: from KS to BA.
  - states: basically the same
  - initial state: just make a unique initial one
  - transition labels: all possible combinations of atomic props
  - states and transitions
  - transitions in  $A$  allowed if
    - covered by accessibility in the KS (+ initial transition added)
    - transition **labelled** by the “post-state-labelling” from KS

# From Kripke Structures to Büchi Automata

## Obtaining the automaton

- Given<sup>3</sup>  $M = (W, R, W_0, V)$
- An  $\omega$ -regular automaton  $A = (S, s_0, L, F, T)$  can be obtained from a Kripke structure as follows

States  $S = W + \{i\}$  and  $s_0 = \{i\}$ , and  $F = W + \{i\}$

Transition labels  $L = 2^{AP}$

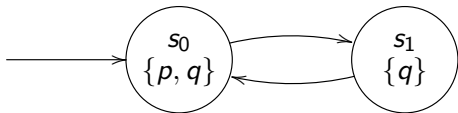
- Transitions
- $s \xrightarrow{l} s' \in T$  iff  $s \rightarrow_M s' \in R \wedge l = V(s')$   
 $s, s' \in S$  s.th.
  - $i \xrightarrow{l} s \in T$  iff  $s \in W_0 \wedge l = V(s)$

---

<sup>3</sup>We call the states here now  $W$  for worlds, to distinguish it from the states of the automaton.

## Example: From Kripke Structures to Büchi Automata

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



The corresponding Büchi Automaton:

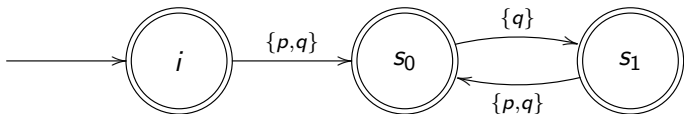


Figure: Büchi-Automaton



# From Logic to Automata

- cf. Regular expressions and FSAs
- For any LTL formula  $\psi$ , there exists a Büchi automaton that accepts precisely those runs for which the formula  $\psi$  is satisfied

stabilization: “eventually always  $p$ ” ( $\diamond\Box p$ ):



We will see the algorithm next lecture . . .

## (Lack of?) expressiveness of LTL

- note: analogy with regular expressions and FSAs: **not 100%**
- in the finite situations: “logical” specification language (regex) correspond fully to machine model (FSA)
- here: LTL is **weaker!** than BAs
- **$\omega$ -regular** expressions +  $\omega$ -regular languages
  - generalization of regular languages
  - allowed to use  $r^\omega$  (not just  $r^*$ )

### Generalization of RE / FSA to infinite words

$\omega$ -regular language correspond to NBAs

- there exist a “crippled” form of “infinite regular expressions” that match LTL (but not relevant here)

- 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
- $\Box X(p)$  certainly does not capture it:



- $p \ \&\& \ \Box(p \rightarrow X!p) \ \&\& \ \Box(!p \rightarrow Xp)$  does not capture it either (because now  $p$  *must* always hold after all even steps):

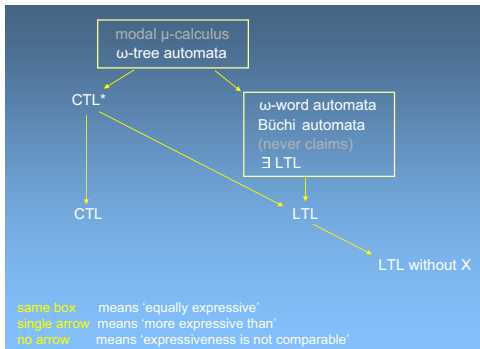


(!t12ba -f)

- $\exists t, t \ \&\& \ \Box(t \rightarrow X!t) \ \&\& \ \Box(!t \rightarrow Xt) \ \&\& \ \Box(!t \rightarrow !p)$   
this formula expresses it correctly



# Expressiveness



- remember: MC checks “system against formula”  $S \models \varphi$

## Linear time approach

$\omega$ -language of the behavior of  $S$  is *contained* in the language allowed by  $\varphi$

- core idea then:<sup>4</sup> instead of

$$\mathcal{L}(S) \subseteq \mathcal{L}(P_\varphi)$$

do the following

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

where  $S$  is a model of the system  $P_\varphi$  represents the property  $\varphi$

---

<sup>4</sup>Compare: refutation in logic.

# What do we need to have to automatically do MC?

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

## Algorithms required for

1. language **emptiness**: are there any accepting runs?
2. language **intersection**: are there any runs accepted by two or more automata?
3. language **complementation**
  - thankfully: all that is **decidable**
  - since operating on Büchi-automata: also **translation** LTL to Büchi needed

## How could one do it, then?

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

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

- **property** is originally given as an LTL formula  $\varphi$
- $\varphi$  is translated into a Büchi automaton  $B_\varphi$ <sup>5</sup>
- And then, as said:

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

One can do better though...

---

<sup>5</sup>Alternatively, the property can be given directly as a Büchi automaton

In practice (e.g., in SPIN): avoid automata **complementation**:

- Assume  $A$  as before
- The **negation** of the property  $\varphi$  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:

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

- If intersection is empty:  $A \models \varphi$ , i.e., “property  $\varphi$  holds for  $A$ ” or “ $A$  satisfies property  $\varphi$ ”
- else:
  - $A \not\models \varphi$
  - **bonus**: accepted word in the intersection **counter example**



# Asynchronous product

- 2 kinds here: synchronous and asynchronous
- asynchronous
  - programs running in parallel
  - interleaving
  - no synchronization!
  - one automaton does something, the others not
- synchronous
  - system together with formula (resp. the automaton representing the formula)
  - lock-step
  - however: stuttering.<sup>6</sup>
- standard (but see terminating condition = definition of final states)

---

<sup>6</sup>Deviating thereby from the classical notion of synchronous product

## Definition (Asynchronous product)

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

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

## 3n+1 inspired example

- 3n+1 problem, long time unsolved problem<sup>7</sup>
- Assume 2 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  $\varphi$  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?

Note: parametrized, infinite problem!

---

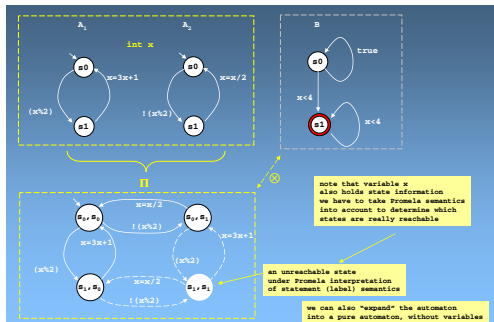
<sup>7</sup>Collatz's problem, Hasse-Collatz problem, Ulam's conjecture, Kakutani's problem, Thwaites conjecture, Hasse's algorithm, or the Syracuse problem.

- guarded commands (thanks to Dijkstra)
- conditional transitions, predicated on a guard
- In Promela semantics, an expression statement has to evaluate to non-zero to be executable ([enabled](#)). 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  $!(\text{false})$  which is  $(\text{true})$ .

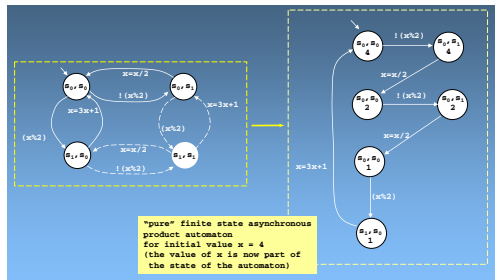
# Example: Async. product

Ignore  $B$  on the right-hand side first



# Example: Pure automaton

initial value:  $x = 4$



- 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)<sup>8</sup>

---

<sup>8</sup>And, in my humble opinion, should then better **not** be called asynchronous product, but synchronous.

## Definition (Stuttering synchronous product)

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  $\varepsilon$  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  $s_2 \in B.F$



# Example: $3n + 1$ system and property

## Example

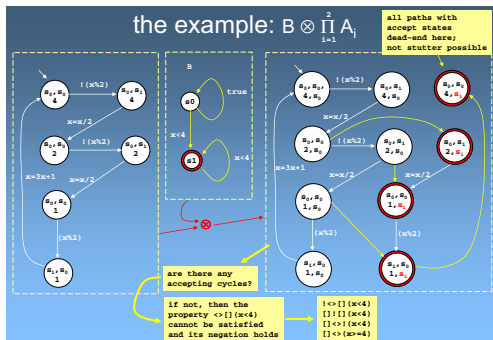


Figure: Synchronous product

- 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 from 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

# Algorithmic checking for emptiness

- For FSA: emptiness checking is easy: **reachability**
- For Büchi:
  - more complex acceptance (namely  $\omega$ -often)
  - simple, one time reachability not enough

⇒ “repeated” reachability

⇒ from initial state, reach an accepting state, and then again, and then again . . .

- Cf. “Lasso” picture
- technically done with the help of SCCs.

## 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

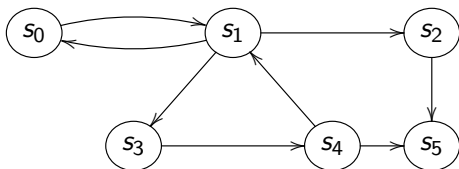


Figure: Strongly connected component

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

## Checking emptiness

Büchi automaton  $A = (S, s_0, L, T, F)$  with accepting run  $\sigma$

### Core observation

Since  $S$  is finite, there is some suffix  $\sigma'$  of  $\sigma$  s.t. every state on  $\sigma'$  is reachable from any other state on  $\sigma'$

- I.a.w: states in  $\sigma'$  are contained in a SCC of the graph of  $A$
- This component is reachable from an initial state and contains an accepting state

### Emptiness check

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

# Emptiness checking and counter example

- different algos for 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  $\sigma_1$  and  $\sigma_2$  are finite sequences



# Model Checking Algorithm

- Let  $A$  be the automaton specifying the system and  $\bar{B}$  the automaton corresponding to the negation of the property  $\varphi$
1. Construct the intersection automaton  $C = A \cap \bar{B}$
  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  $\varphi$
  4. Otherwise,
    - 4.1 Take one strongly-connected component  $SC$  of  $C$
    - 4.2 Construct a path  $\sigma_1$  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  $\sigma_2$  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  $\varphi$

## 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

- 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:** 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

- translation to Generalized Büchi GBA
- cf. Thompson's construction
- *structural* translation
- Crucial idea: connect semantics to the syntax.
- compare Hintikka-sets or similar constructions for FOL

- transition systems TS:
  - corresponds to Kripke systems
  - state-labelled<sup>9</sup>
  - labelled by sets of atomic props:  $\Sigma = 2^{AP}$
  - “language” or behavior of the TS: (traces): infinite sequences over  $\Sigma$

---

<sup>9</sup>transition labels irrelevant

## Illustrative examples (5.32)

1.  $\square \diamond green$
2.  $\square (request \rightarrow \diamond response)$
3.  $\diamond \square a$



- equi-expressive than NBA
- used in the construction
- different way of defining acceptance
- Acceptance: set of **acceptance sets** = set of sets of elements of  $Q$ .
- Acceptance: each acceptance set  $F_i$  must be “hit” infinitely often

## Basic idea for $\mathcal{G}_\varphi$

- not the construction yet, but: “insightful” property
- find a mental picture:
  - what are the states of the automaton
  - (and how are they connected by transitions)
- $A_i \in \Sigma$ , sets of atomic props
- $B_i$  : “extended”, i.e.,  $A_i \supseteq B_i$ .

### States as sets of formulas

Namely those that are intended to be in the “language of that state”. I.e., the  $B_i$ ’s form the states of  $\mathcal{G}_\varphi$ .

Intentions: given  $\sigma = A_0A_1A_2 \dots \in \mathcal{L}(\varphi)$

$$\psi \in B_i \quad \text{iff} \quad A_i, A_{i+1}A_{i+2} \dots \models \psi$$

- related to Fisher-Ladner closure
- See page 276
- what's a “closure” in general
- Extending  $A_i$  to  $B_i$  not by *all* true formulas, but only those that could **conceivably play a role** in an automaton checking  $\varphi$
- $\Rightarrow$ : achieving “finiteness” of the construction

## How to extend $A_i$ 's

- not by irrelevant stuff (closure of  $\varphi$ ).
- two other conditions:
  - avoid contradictions (**consistency**)
  - include logical consequences<sup>10</sup> (**maximality**)
- maximally consistent sets! (here called *elementary*)
- in one state: local perspective only (but don't forget  $U$ )
- Cf: KS has an interpretation for each  $AP$ , here now (in the intended BA),

“semantics” (states) by “syntax”

“interpretation” for *all relevant formulas* “in” each state  
(subformulas of  $\varphi$  and their negation)

---

<sup>10</sup>hence the notion of “closure”

- given  $\varphi$
- **elementary**: “maximally consistent set of subsets (of the closure of  $\varphi$ )”
- **consistent**: “no obvious contradictions”
  - wrt. prop logic
  - *locally consistent* wrt. until
- “maximal”

Example:  $\varphi = aU(\neg a \wedge b)$

# Construction of GNBA: general

- given  $AP$  and  $\varphi$
- given  $\varphi$ , construct an GNBA such that

$$\mathcal{L}(B) = \text{words}(\varphi)$$

- 3 core ingredients
  1. states = sets of formulas which “hold” in that state
  2. transition relation: connect the states appropriately,
  3. transitions labelled by sets of  $AP$ .
  4. outgoing transitions in the BA “enabled” “with transitions labelled by  $2^{AP}$ , like

if  $\bigcirc\psi \in$  a state and  
if

- [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.