Chapter 1

LTL model checking

Course "Model checking"

Volker Stolz, Martin Steffen

Autumn 2019

Section

Logic model checking: What is it about?

The basic method
General remarks
Motivating examples

Chapter 1 “LTL model checking”
Course “Model checking”
Volker Stolz, Martin Steffen
Autumn 2019

Logic model checking (1)

- a technique for verifying *finite-state* (concurrent) systems

Often involves steps as follows

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

State-space explosion

Main causes of combinatorial complexity in SPIN/Promela (and in other model checkers.)

- The number of and size of buffered channels
- The number of asynchronous processes

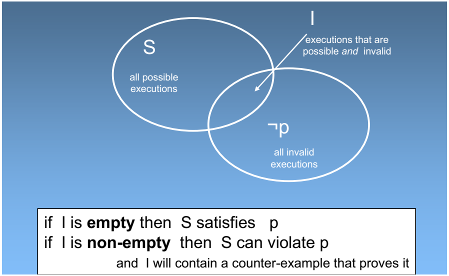
The basic method

- 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

The basic method

- 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

The basic method



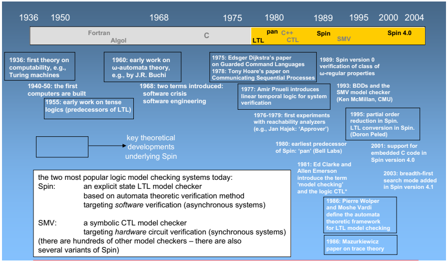
Scope of the method

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
- Incompleteness of specification
- Dead code {(unreachable code)}
- Violation of certain system bounds
- Logic problems: e.g, temporal relations
- ...

A bit of history



On correctness (reminder)

- A system is correct if it meets its design requirements.
- There is no notion of "absolute" correctness: It is always wrt. 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

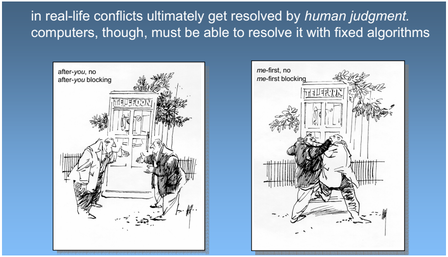
- The use of abstraction is needed for building models (systems may be extremely big)
  - A model is always an abstraction of the reality
- The choice of the model/abstractions depends on the requirements to be checked
- A good model keeps only relevant information
  - A trade-off must be found: too 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



A Small multi-threaded program

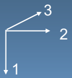
```
int   x,  y,  r;
int  *p, *q, *z;
int **a;

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

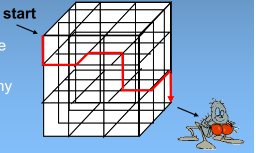
- the number of possible thread interleavings is...



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

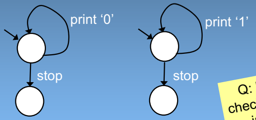
placing 3 sets of 3 tokens in 9 slots

- 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?

Section

Automata and logic

- Finite state automata
- Büchi Automata
- Something on logic and automata
- Implications for model checking

Chapter 1 "LTL model checking"
Course "Model checking"
Volker Stolz, Martin Steffen
Autumn 2019

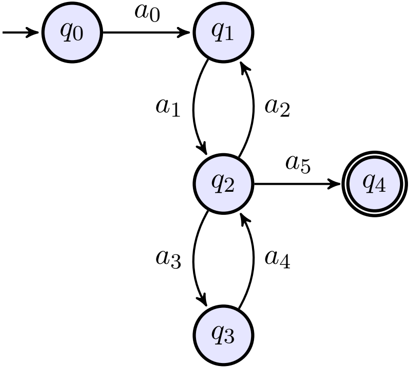
FSA

Definition (Finite-state automaton)

A *finite-state automaton* is a tuple $(Q, q_0, , \Sigma, F, \rightarrow)$, where

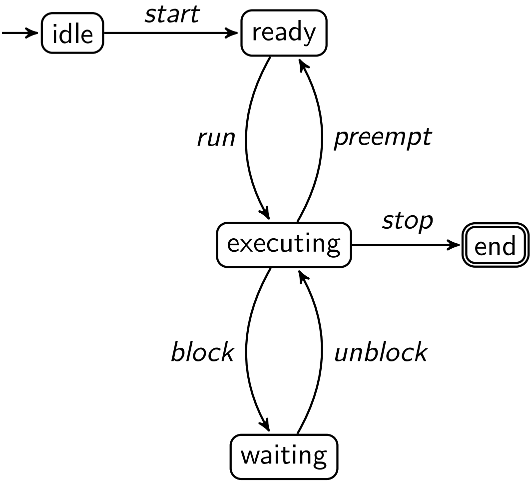
- Q is finite set of states
- $q_0 \in Q$ is a distinguished initial state
- the "alphabet" Σ is a finite set of labels (symbols)
- $F \subseteq Q$ is the (possibly empty) set of final states
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, connecting states in Q.

Example FSA



Example: An interpretation

The above automaton may be interpreted as a *process scheduler*:



Determinism vs. non-determinism

Definition (Determinism)

A finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is deterministic iff

$$q_0 \xrightarrow{a} q_1 \ \wedge q_0 \xrightarrow{a} q_2 \implies q_1 = q_2$$

Runs

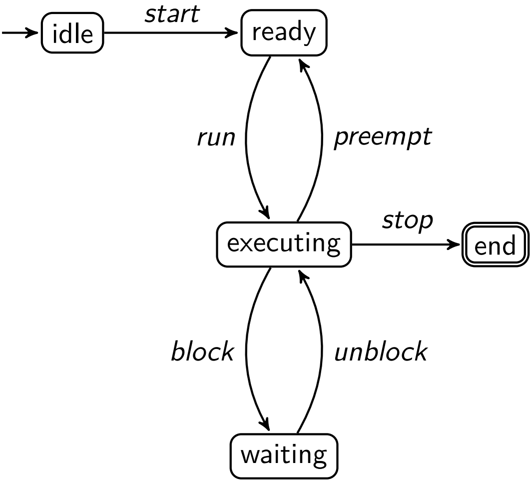
Definition (Run)

A *run* of a finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is a (possibly infinite) sequence

$$\sigma = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \dots$$

- $q \xrightarrow{a} q'$ is meant as $(q, a, q') \in \rightarrow$
- each run corresponds to a state sequence (a word) over Q and a word over Σ

Example run

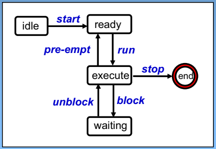


- state sequences from runs: idle ready $(\text{execute waiting})^*$
- corresponding words in Σ: $start\ run(block, unblock)^*$
- A single state sequence may correspond to more than one word

“Traditional” acceptance

Definition (Acceptance)

An accepting run of a finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is a finite run $\sigma = q_0 \overset{a_0}{\rightarrow} q_1 \overset{a_1}{\rightarrow} \ldots \overset{a_{n-1}}{\rightarrow} q_n$, with $q_n \in F$



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

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

Accepted language

Definition (Language)

The *language* $\mathcal{L}(\mathcal{A})$ of automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is the set of words over Σ that correspond to the set of all the accepting runs of $\mathcal{A}$.

- generally: *infinitely* many words in a language
- remember: regular expressions etc.

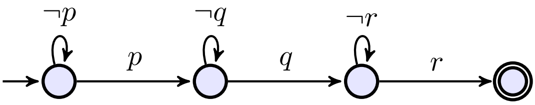
Reasoning about runs

Sample correctness claim (positive formulation)

If first p becomes true and afterwards q becomes true, then afterwards, r can no longer become true

Seen negatively

It's an error if in a run, one sees first p, then q, and then r.



- reaching accepting state $\Rightarrow$ correctness property violation
- accepting state represents error

Comparison to FSA in “standard” language theory

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

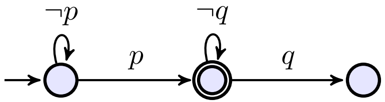
Reasoning about infinite runs

Some liveness property

"if p then eventually q."

Seen negatively

It's an error if one sees p and afterwards never q (i.e., forever $\neg q$)



- violation: only possible in an infinite run
- not expressible by *conventional* notion of acceptance

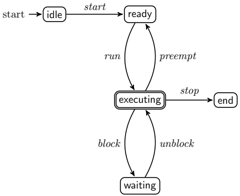
Büchi acceptance

- infinite run: often called ω-*run* ("omega run")
- corresponding acceptance properties: ω-acceptance
- different versions
  - The so-called Büchi, Muller, Rabin, Streett, etc., acceptance conditions
  - Here, for now: Büchi acceptance condition [3] [2]

Definition (Büchi acceptance)

An accepting ω-run of finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is an infinite run σ such that some $q_i \in F$ occurs infinitely often in σ

Example: "process scheduler"



- accepting ω-runs
- ω-language

infinite state sequence	ω-word
idle (ready executing)$^\omega$	*start* (*run preempt*)$^\omega$

Generalized Büchi automata

Definition (Generalized Büchi automaton)

A *generalized Büchi automaton* is an automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$, where $F \subseteq 2^Q$.
Let $F = \{f_1, \ldots, f_n\}$ and $f_i \subseteq Q$. A run σ of $\mathcal{A}$ is *accepting* if

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

- $inf(\sigma)$: states visited infinitely often in σ
- generalized Büchi automaton: multiple accepting sets instead of only one ($\neq$ "original" Büchi Automata)
- generalized Büchi automata: *equally* expressive

Stuttering

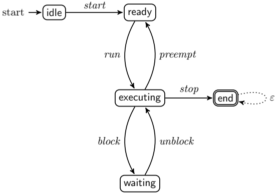
- treat finite and infinite acceptance uniformely
- finite runs as inifite ones, where, at some point, infinitely often "nothing" happens (stuttering)
  - Let ε be a predefined nil symbol
  - alphabet/label set extended to $\Sigma + \{\varepsilon\}$
  - extend a finite run to an equivalent infinite run: keep on stuttering after the end of run. The run must end in a final state.

Definition (Stutter extension)

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

$$\sigma \ (s_n, \varepsilon, s_n)^\omega \tag{1}$$

Stuttering example



IN5110 – Verification and specification of parallel systems

From Kripke structures to Büchi automata

- 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 unrolling of the Kripke structure.

Kripke structure (reminder)

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
- $V : S \to 2^{AP}$ is a map labeling each state with a set of propositional variables

Notation: $\to$ for accessibility relation
A path in M is an infinite sequence $\sigma = s_0, s_1, s_2, \ldots$ of states such that $s_i \to s_{i+1}$ (for all $i \geq 0$).

BAs vs. KSs

- "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 $\mathcal{A}$ allowed if
    - covered by accesssibility in the KS (+ initial transition added)
    - transition labelled by the "post-state-labelling" from KS

KS to BA

Given $M = (W, R, W_0, V)$. An automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ can be obtained from a Kripke structure as follows

transition labels: $\Sigma = 2^{AP}$

states:

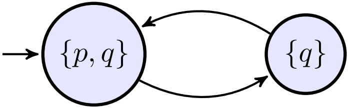
- $Q = W + \{i\}$
- $q_0 = i$
- $F = W + \{i\}$

transitions:

- $s \xrightarrow{a} s'$ iff $s \rightarrow_M s'$ and $a = V(s')$ $s, s' \in W$
- $i \xrightarrow{a} s \in T$ iff $s \in W_0$ and $a = V(s)$

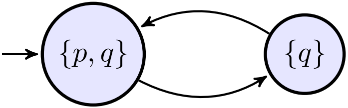
Example: KS to BA

A Kripke structure (whose only infinite run satisfies (for instance) $\Box q$ and $\Box\Diamond p$):

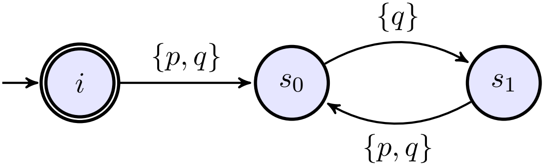


Example: KS to BA

A Kripke structure (whose only infinite run satisfies (for instance) $\Box q$ and $\Box\Diamond p$):



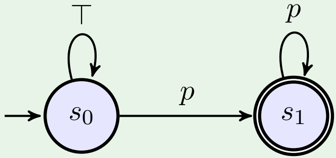
The corresponding Büchi automaton:



From logic to automata

- cf. regular expressions and FSAs
- for any LTL formula φ, there exists a Büchi automaton that accepts precisely those runs for which the formula φ is satisfied

Example (stabilization: "eventually always p", $\Diamond\Box p$:)



(Lack of?) expressiveness of LTL

- note: analogy with regular expressions and FSAs: not 100 percent
- in the finite situation: "logical" specification language (regexp) correspond fully to machine model (FSA)
- here: LTL is weaker! than BAs
- ω-regular expressions + ω-regular languages
- generalization of regular languages
- allowed to use r^ω (not just r^*)

Generalization of RE / FSA to infinite words

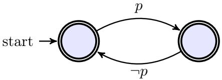
ω-regular language correspond to NBAs

ω-regular properties strictly more expressive than LTL

Temporal property

p is always false after an *odd* number of steps

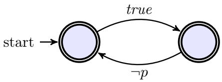
$$p \wedge \Box(p \rightarrow \bigcirc\neg p) \wedge \Box(\neg p \rightarrow \bigcirc p)$$



ω-regular properties strictly more expressive than LTL

Temporal property

p is always false after an *odd* number of steps

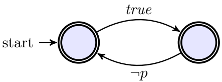


ω-regular properties strictly more expressive than LTL

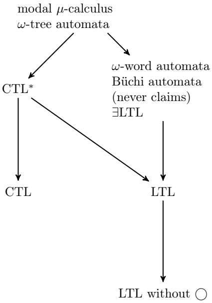
Temporal property

p is always false after an *odd* number of steps

$$\exists t.\ t \land \Box(t \to \bigcirc\neg t) \land \Box(\neg t \to \bigcirc t) \land \Box(\neg t \to p)$$



Expressiveness



Core part of automata-based MC

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

Linear time approach

- ω-language of the behavior of S is *contained* in the language allowed by φ

- core idea then: instead of

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

do

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

where S is a model of the system P_φ represents the property φ

What's needed for automatic MC?

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

Algorithms needed for

1. translation LTLto Büchi
2. language emptiness: are there any accepting runs?
3. language intersection: are there any runs accepted by two or more automata?
4. language complementation

- thankfully: all that is *decidable*

How could one do it, then?

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

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

- *property* originally given as an LTL formula φ
- translate φ into a Büchi automaton B_φ
- check

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

How could one do it, then?

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

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

- *property* originally given as an LTL formula φ
- translate φ into a Büchi automaton B_φ
- check

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

One can do better though. . .

One can do better

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

- Assume A as before
- The negation of the property φ 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 φ holds for A" or "A satisfies property φ"
- else:
  - $A \not\models \varphi$
  - bonus: accepted word in the intersection counter example

Section

Automata products

Chapter 1 “LTL model checking”
Course “Model checking”
Volker Stolz, Martin Steffen
Autumn 2019

Two kinds of product

- conceptually standard (but see terminating condition = definition of final states)

asynchronous

- prog's running in parallel
- interleaving
- no synchronization!
- one automaton does something, the others not

synchronous

- together with (the automaton representing) the formula
- lock-step
- however: *stuttering*.

Asychronous product

Definition (Asynchronous product)

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 $\forall j, 1 \leq j \leq n, j \neq i \implies (x_j = y_j)$
- $A.F$ contains those states from $A.S$ that satisfy $\forall (A_1.s, A_2.s, \ldots, A_n.s) \in A.F, \exists i, 1 \leq i \leq n, A_i.s \in A_i.F$

3n+1 inspired example

- 3n+1 problem
- 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 $x/2$

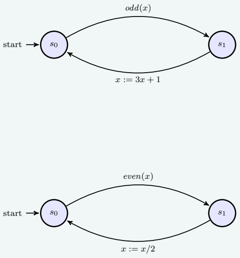
Question

Does the corresponding function *terminate* for all inputs x?

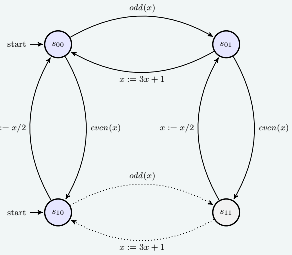
- Let φ the following property: $\Box\Diamond(x \geq 4)$ (negated $\Diamond\Box(x < 4)$)

Example: async product

A_1 and A_2



$A_1 \times A_2$



Tests or guards on transitions

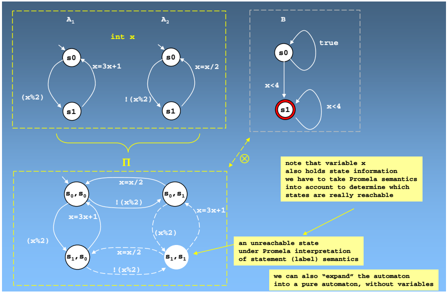
- guarded commands (thanks to Dijsktra)
- conditional transitions, predicated on a guard
- in Promela[1] semantics, an expression statement has to evaluate to non-zero to be executable (endabled). So to test whether a variable `x` is odd, we write `!(x%2)`, and `(x%2)` for checking whether `x` is even.

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

[1] Probably inspired by C . . .

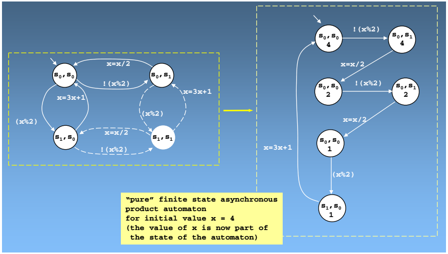
Example: Async. product

- ignore B on the right-hand side first
- final states not really important



Example: Pure automaton

initial value: $x = 4$



Remarks

- Not all the states in the product necessarilty reachable from q_0

[2] And, in my humble opinion, should then better not be called asynchronous product, but synchronous.

Remarks

- Not all the states in the product necessarilty reachable from q_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)

[2] And, in my humble opinion, should then better not be called asynchronous product, but synchronous.

Remarks

- Not all the states in the product necessarilty reachable from q_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 asyc. product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time $\Rightarrow$ interleaving

[2] And, in my humble opinion, should then better not be called asynchronous product, but synchronous.

Remarks

- Not all the states in the product necessarilty reachable from q_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 asyc. product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time $\Rightarrow$ interleaving
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)

[2] And, in my humble opinion, should then better not be called asynchronous product, but synchronous.

Remarks

- Not all the states in the product necessarilty reachable from q_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 asyc. product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time $\Rightarrow$ interleaving
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)
  - Some transitions may synchronize by sending and receiving a message

[2] And, in my humble opinion, should then better not be called asynchronous product, but synchronous.

Remarks

- Not all the states in the product necessarilty reachable from q_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 asyc. product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time $\Rightarrow$ interleaving
- 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)[2]

[2] And, in my humble opinion, should then better not be called asynchronous product, but synchronous.

Synchronous product

Definition (Synchonous product)

The synchronous product of two finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$ (written $\mathcal{A}_1 \otimes \mathcal{A}_2$ is defined as finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ where:

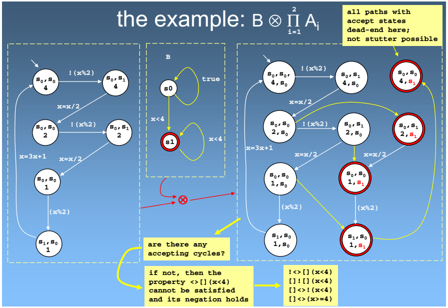
- $Q = Q_1 \times Q_1$
- $q_0 = (q_{01}, q_{02})$
- $\Sigma = \Sigma_1 \times \Sigma_2$.
- $\rightarrow = \rightarrow_1 \times \rightarrow_2$
- $(q_1, q_2) \in F$ if $q_1 \in F_1$ or $q_2 \in F_2$

- The latter condition: not so important in general, resp. up-to debate
- Synchonous product here:
  - used in connection with stuttering
  - for LTL to Büchi

Let the system automaton stutter

- asymmetric situation
- one automaton: “system”
- second one:
  - “recognizer”
  - automaton that represents the logical LTL formula
- for system automating: add stuttering
- stutter: a self-loop labeled with ε at every every state in without outgoing transitions

Example: synch. product for $3n + 1$ system and property



Remarks

- We require the stutter-closure of P (as 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

Remarks

- We require the stutter-closure of P (as 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 states necessarily reachable from q_0

Remarks

- We require the stutter-closure of P (as 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 states necessarily reachable from q_0
- Main difference between asynchronous and synchronous products: labels and transitions. for synchronous product:
    - *joint* transitions of the component automata
    - labels are pairs: the combination of the two labels of the original transitions in the component automata

Remarks

- We require the stutter-closure of P (as 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 states necessarily reachable from q_0
- Main difference between asynchronous and synchronous products: labels and transitions. for synchronous product:
  - *joint* transitions of the component automata
  - labels are pairs: the combination of the two labels of the original transitions in the component automata
- In general *here* $P \otimes B \not\equiv B \otimes P$, but given that in SPIN B is particular kind of automaton (labels are state properties, not actions), we have then $P \otimes B \equiv B \otimes P$

Section

Model checking algorithm

Chapter 1 “LTL model checking”
Course “Model checking”
Volker Stolz, Martin Steffen
Autumn 2019

Algorithmic checking for emptyness

- for FSA: emptyness checking is easy: reachability
- For Büchi:
  - more complex acceptence (namely ω-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.

Strongly-connected components

Definition (SCC)

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

SCC example

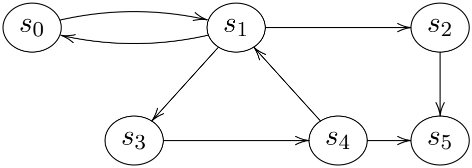


Figure: Strongly connected component

SCC example

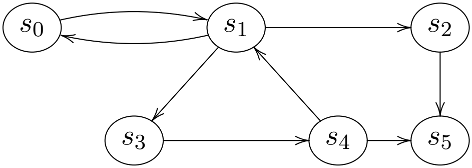


Figure: Strongly connected component

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

SCC example

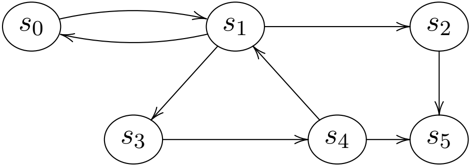


Figure: Strongly connected component

- 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:

SCC example

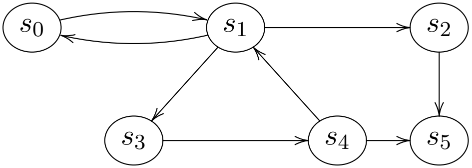


Figure: Strongly connected component

- 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 $A = (Q, s_0, \Sigma, \rightarrow, F)$ with accepting run σ

Core observation

As Q is finite, there is some suffix σ' of σ s.t. every state on σ' is reachable from any other state on σ'

- I.a.w: those set of states is strongly connected.
- This set is reachable from an initial state and contains an accepting state

Emptyness 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

Emptyness 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

Emptyness 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

Emptyness 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 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 φ

Model checking algorithm

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

1. Construct the intersection automaton $C = A \cap \overline{B}$

Model checking algorithm

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

1. Construct the intersection automaton $C = A \cap \overline{B}$
2. Apply an algorithm to find SCCs reachable from the initial states of C

Model checking algorithm

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

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 φ

Model checking algorithm

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

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 φ
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 φ

LTL to Büchi

- 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

Source and terminology: Baier and Katoen [1]

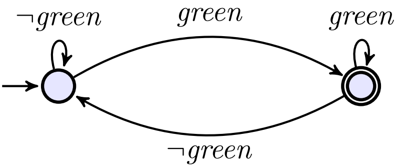
- transition systems TS:
  - corresponds to Kripke systems
  - state-labelled[3]
  - labelled by sets of atomic props: $\Sigma = 2^{AP}$
  - "language" or behavior of the TS: (traces): infinite sequences over Σ

[3] transition labels irrelevant

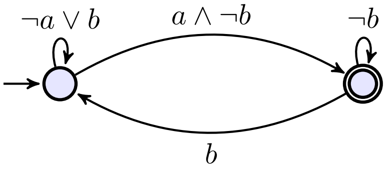
Illustrative examples (5.32)

1. $\Box\Diamond green$
2. $\Box(request \rightarrow \Diamond response)$
3. $\Diamond\Box a$

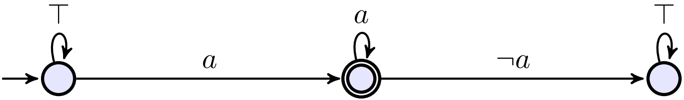
$\Box\Diamond green$



$\Box(request \rightarrow \Diamond response)$



$\Diamond\Box a$



Reminder: Generalized NBA

- 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" (by sub-formulas of φ), i.e., $B_i \supseteq A_i$.

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 $\mathcal{G}_\varphi$.

Given $\sigma = A_0 A_1 A_2 \ldots \in \mathcal{L}(\varphi)$.
Extension to $\hat{\sigma} = B_0 B_1 B_2 \ldots$

$$\psi \in B_i \qquad \text{iff} \qquad \underbrace{A_i, A_{i+1} A_{i+2} \ldots}_{\sigma^i} \models \psi$$

$\hat{\sigma}$ = run (state-sequence) in $\mathcal{G}_\varphi$

Cf. FSAs

- states as “sets” of “words” (language resp. set of ltl formulas)
- cf. Myhill-Nerode
- a bit different, (equivalence on languages of finite words)
- represent states by equivence classes of words

Closure of φ

- related to Fisher-Ladner closure
- See page 276
- "states" A_i from the mental picture
- 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 φ
- $\Rightarrow$ achieving "finiteness" of the construction

How to extend A_i's

- not by irrelevant stuff (closure of φ).
- two other conditions:
  - avoid contradictions (consistency)
  - include logical consequences[4] (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 φ and their negation)

[4] hence the notion of "closure"

Elementary sets/maximally consistent sets

- given φ
- elementary: "maximally consistent set of subsets (of the closure of φ)"
- consistent: "no obvious contradictions"
- maximally consistent: sets for formulas ψ in the closure of φ s.t., there exists some path π s.t. $\pi \models \psi$.
  - wrt. propositional logic
  - *locally consistent* wrt. until
- "maximal"

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

$$\{a, b, \varphi\} \subseteq closure(\varphi)$$

$$\{\neg a, \neg b, \neg(\neg a \wedge b), \varphi\}?$$

page 276/277

Construction of GNBA: general

- given AP and φ
- given φ, construct an GNBA such that

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

- 3 core ingredients
  1. states = sets of formulas which (are suppsed to) "hold" in that state
  2. transition relation: connect the states appropriately,
  3. transitions labelled by sets of AP.
- labeled transition connected states to match the semantics: for $\bigcirc\varphi$:

simplified for $\bigcirc$

go from a state containing $\bigcirc\varphi$ to a state containing φ. Label the transition with the APs from the start state.

Transition relation

$$\delta : Q \times 2^{AP} \to 2^Q$$

- if $A \neq B \cap AP$: $\delta(B, A) = \emptyset$
- if $A = B \cap AP$, then $\delta(B, A)$ is the set B' such that
  - for every $\bigcirc\psi \in closure(\varphi)$:

$$\bigcirc\psi \in B \quad \text{iff} \quad \psi \in B'$$

  - for every $\varphi_1\ U\ \varphi_2 \in closure(\varphi)$:

$$\varphi_1\ U\ \varphi_2 \in B \quad \text{iff} \quad \varphi_2 \in B \quad \text{or} \quad (\varphi_1 \in B \quad \text{and} \quad \varphi_1\ U\ \varphi_2 \in B')$$

Section

References

References I

Bibliography

[1] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

[2] Büchi, J. R. (1960). Weak second-order arithmentic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92.

[3] Büchi, J. R. (1962). On a decision method in restricted second-order logic. In *Proceedings of the 1960 Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press.