Model checking: an introduction

FMSS, 2020-2021

Model checking

A fully automated method for analysing properties of systems. Closer to program verification than to program analysis.

A verification technique based on:

- model of the program/system described by a transition system with additional information assigned to states
- computation tree logic for expressing properties as a logic formula
- automatic check that the model satisfies the formula

E.M. Clarke, E.A. Emerson and A.P. Sistla, Automatic verification of finite state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, 1986.

Transition System

In general, we can assume a set of *atomic propositions* and states are assigned a subset of the propositions.

A transition system TS is a tuple $(S, I, \rightarrow, AP, L)$ such that:

- S is a non-empty set of states;
- $I \subseteq S$ is a non-empty set of initial states;
- $\rightarrow \subseteq S \times S$ is the transition relation;
- AP is a set of atomic propositions;
- $L : s \Rightarrow PowerSet(AP)$ is a labelling function for states.
  We write
  $s \rightarrow s'$ if there is a transition from state s to state s'.

CTL formula ϕ

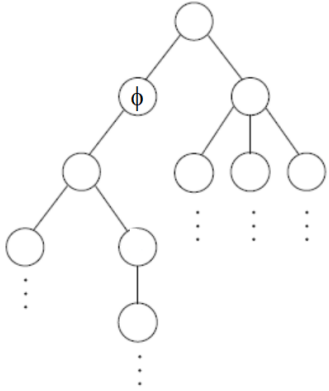
Atomic proposition
$s \models \phi$
state *s* satisfies ϕ, i.e., state *s* is a model of ϕ

The simplest formula is an atomic proposition.
Example: assume $\phi = a$. $s \models \phi$ (*a* holds in s)

Formulae are evaluated in the initial state.

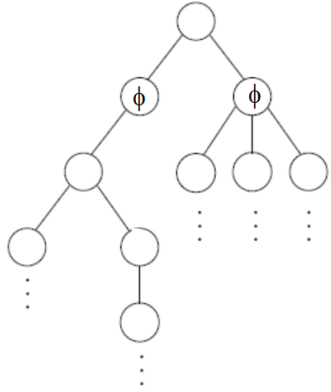
Reachability in one step

$EX\phi$ it is possible in one step to reach a state that satifies ϕ



Reachability in one step

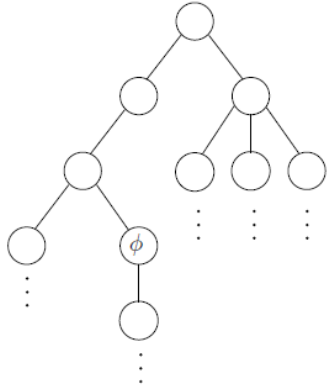
$AX\phi$ the next state it is certain that satifies ϕ



Reachability

EF ϕ

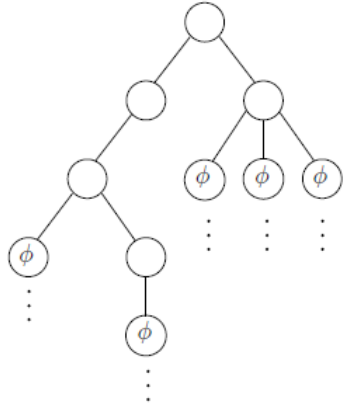
There exists a path and a state along that path such that ϕ holds



Reachability

AF ϕ

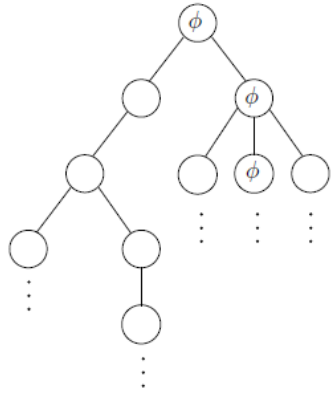
Along every path there exists a state such that ϕ holds at that state



Unavoidability

EG ϕ

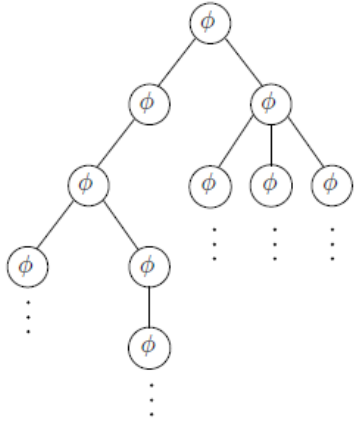
There exists a path such that ϕ holds at every state along that path



Unavoidability

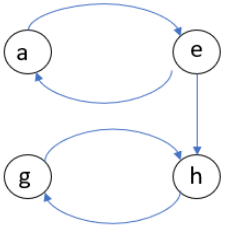
AG ϕ

Along every path ϕ holds at every state



An example

$S = \{a, e, g, h\}$ $I = \{a\}$ $AP = \{consonant, vowel\}$
$L(a) = \{vowel\}$, $L(e) = \{vowel\}$
$L(g) = \{consonant\}$, $L(h) = \{consonant\}$



Atomic proposition

$g \models consonant$ $a \not\models consonant$

Reachability in one step

$e \models EX\ consonant$ $a \not\models EX\ consonant$
$a \models AX\ vowel$ $e \not\models AX\ vowel$

Reachability

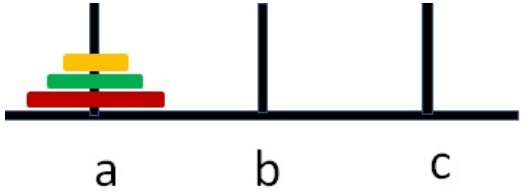
$a \models EF\ consonant$ $g \not\models EF\ vowel$
$a \models AF\ vowel$ $e \not\models AF\ vowel$

Unavoidability

$a \models EG\ vowel$ $g \not\models EG\ vowel$
$g \models AG\ consonant$ $a \not\models AG\ vowel$

Another example of Transition System

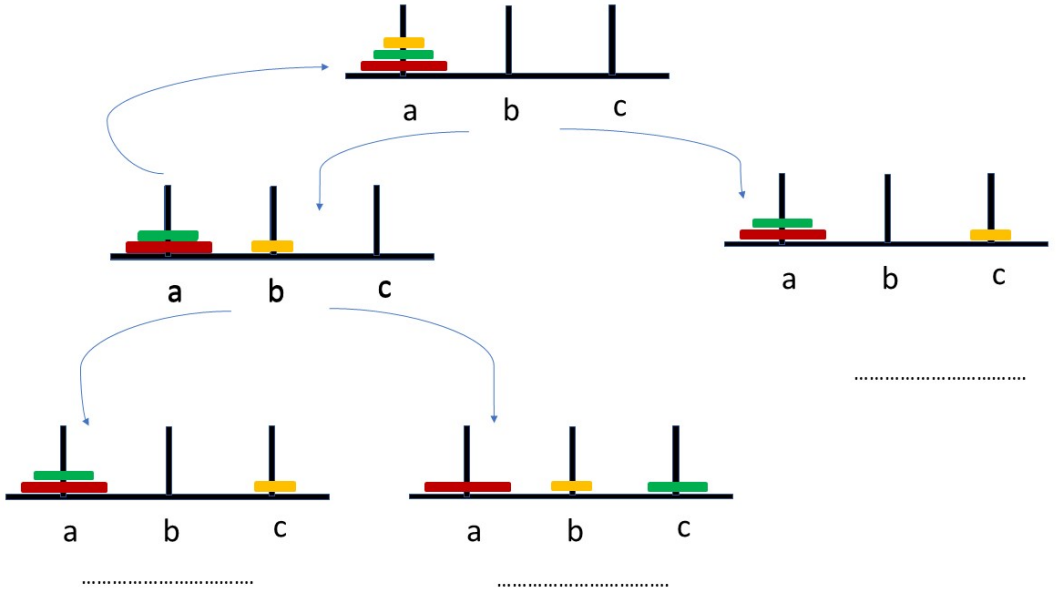
Towers of Hanoi



- Three rods (a, b, c) and three disks with different size (small, medium, large) in order on rod a. The largest disk at the bottom of a.
- Move the entire stack of disks from rod a to rod c, assuming the following rules:
  - only one disk can be moved at a time
  - no larger disk may be placed on top of a smaller disk

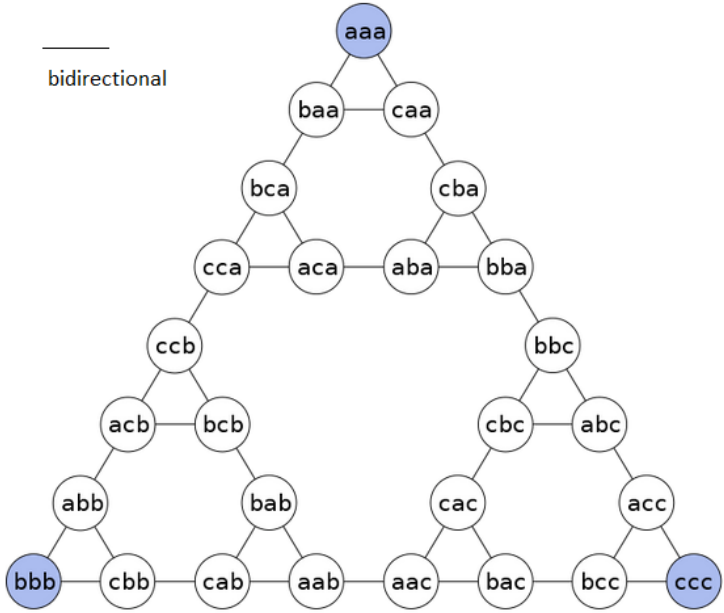
https://en.wikipedia.org/wiki/Tower_of_Hanoi

Hanoi Tower: steps



Transition System

State: sequence of rods on which disks are palced (pos_{small} pos_{medium} pos_{large})



Transition System

$TS = (S, I, \rightarrow, AP, L)$ where:
- S is the set of sequences of three letters chosen among a, b, c;
- $I = \{aaa\}$;
- $\rightarrow$ is the transition relation; the relation is symmetric (in the figure an undirected line)
- $AP = S$, the atomic propositions are chosen the same as the set of states;
- $L(\sigma) = \sigma$, the labelling function is trivial.

We can check if along every path it is always possible to reach the configuration "all disks on rod c". Let $\phi = ccc$.

$$aaa \models AF\ \phi$$

CTL

STATE FORMULAE

$$\phi ::= tt \mid ap \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid E\Psi \mid A\Psi$$

PATH FORMULAE

$$\Psi ::= X\phi \mid F\phi \mid G\phi \mid \phi_1 U \phi_2$$

$\phi_1 U \phi_2$
path formula which requires that exists a state s such that ϕ_2 holds and ϕ_1 holds in all states up to the state s

Other formulae

ff for $\neg tt$
$\phi_1 \vee \phi_2$ for $\neg((\neg\phi_1) \wedge (\neg\phi_2))$
$\phi_1 \implies \phi_2$ for $((\neg\phi_1) \vee \phi_2)$

The meaning of state formulae and path formulae depend on each other.

Semantics of state formulae

$$
\begin{array}{lll}
\sigma \models \mathit{tt} & \text{iff} & \text{true} \\
\sigma \models \mathit{ap} & \text{iff} & \text{ap} \in L(\sigma) \\
\sigma \models \phi_1 \wedge \phi_2 & \text{iff} & (\sigma \models \phi_1) \wedge (\sigma \models \phi_2) \\
\sigma \models \neg\phi & \text{iff} & \sigma \not\models \phi \\
\sigma \models E\Psi & \text{iff} & \exists \pi : \pi \in \mathit{Path}(\sigma) \wedge \pi \models \Psi \\
\sigma \models A\Psi & \text{iff} & \forall \pi : \pi \in \mathit{Path}(\sigma) \implies \pi \models \Psi
\end{array}
$$

E and A have the same meaning as in predicate logic but they range over paths rather than states.

Semantics of path formulae

$$
\begin{array}{lll}
\sigma_0\sigma_1\cdots\sigma_n\cdots \models X\phi & \text{iff} & \sigma_1 \models \phi \wedge n > 0 \\
\sigma_0\sigma_1\cdots\sigma_n\cdots \models F\phi & \text{iff} & \sigma_n \models \phi \wedge n \geq 0 \\
\sigma_0\sigma_1\cdots\sigma_n\cdots \models G\phi & \text{iff} & \forall i : \sigma_i \models \phi \\
\sigma_0\sigma_1\cdots\sigma_n\cdots \models \phi_1 \bigcup \phi_2 & \text{iff} & ((\sigma_n \models \phi_2 \wedge n \geq 0) \\
 & & \wedge(\forall i \in \{0, , n-1\} : \sigma_i \models \phi_1))
\end{array}
$$

For example, $F\ \phi$ holds on a path, whenever ϕ holds on some state of the path, possibly the current state

Definitions

- a state formula ϕ holds on a TS whenever it holds for all initial states: $\forall \sigma \in I : \sigma \models \phi$
- A path in a transition system is a sequence of states $\sigma_0\sigma_1 \cdots \sigma_{n-1}\sigma_n \cdots$, where $\forall n > 0, \sigma_{n-1} \rightarrow \sigma_n$, and where the path is as long as possible.
  $Path(\sigma_0)$ denotes the set of paths $\pi = \sigma_0\sigma_1 \cdots \sigma_{n-1}\sigma_n \cdots$ starting in σ_0. .
- state σ is *stuck* if there are no transitions leaving σ. We have $Path(\sigma) = \sigma$.

Definitions

- Given $S_0 \subseteq S$, $Reach_1(S_0) = \{\sigma_1 \mid \sigma_0\sigma_1 \cdots \sigma_n \cdots \in Path(\sigma_0)$ and $\sigma_0 \in S_0\}$
  states reachable from a state in S_0 in one step.
- Given $S_0 \subseteq S$, $Reach(S_0) = \{\sigma_n \mid \sigma_0\sigma_1 \cdots \sigma_n \cdots \in Path(\sigma_0)$ and $\sigma_0 \in S_0$ and $n \geq 0\}$
  states reachable from a state in S_0 in zero or more steps.
- $Reach(I)$ is the set of reachable states.

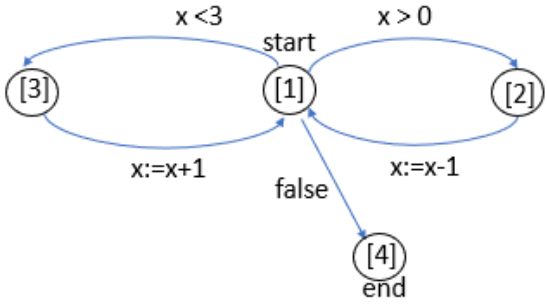
Analysis of programs

Program = Program Graph + Data

- Program graph: represents the control structure of the program.
- memory: represents the data structure on which the program operates.
- the semantics of the program is based on the memories at the different program points.
  When we execute an instruction we move from a pair (pp, m) to another pair (pp', m').
  The value of pp' and m' depends on the values pp and m and the semantics of the instruction that is executed.

Analysis of programs

An example



Assume:
Program point *pp*: [1], [2], [3] and [4], with [1] the initial point
x can take value : 0, 1, 2, 3
Memory *m*: $(x, 0)$,$(x, 1)$, $(x, 2)$ and $(x, 3)$.

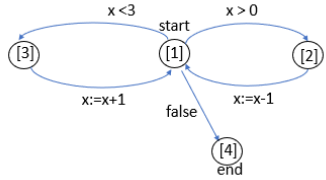
Transition System TS

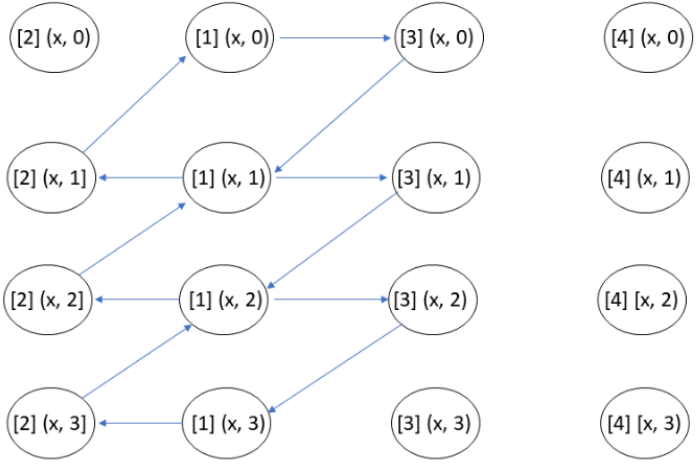
- $S = \{(pp, m) \mid pp \in \{[i], i = 1, 2, 3, 4\} \wedge$
  $m \in \{(x, i), i = 0, 1, 2, 3\}\}$
- $I = \{([1], (x, i)), i = 0, 1, 2, 3\}$
- $\rightarrow$ .....
- $AP = \{@[i], i = 1, 2, 3, 4\} \cup \{@(x, i), i = 0, 1, 2, 3\} \cup \{start\} \cup \{end\}$
- $L([1], (x, i)) = \{@1, @(x, i), start\}$
  $L([2], (x, i)) = \{@2, @(x, i)\}$
  $L([3], (x, i)) = \{@3, @(x, i)\}$
  $L([4], (x, i)) = \{@4, @(x, i), end\}$

Each state is labelled by the program point and the memory it consists of, and we have explicit labels for the initial and final program points.

Transition System TS

$L([1],(x,i)) = \{@1, @(x,i), start\}$, $L([2],(x,i)) = \{@2, @(x,i)$
$L([3],(x,i)) = \{@3, @(x,i)\}$, $L([4],(x,i)) = \{@4, @(x,i), end\}$





How many states? How many reachable states?

Consider the set of states where the following formula holds:
@[3]
@$(x, 2)$
@$[1] \wedge$ @$(x, 2)$

Termination of the system

$$start \implies EF\ end$$

for each initial state it is possible to terminate

$$start \implies AF\ end$$

for each initial state it is certain to terminate

When the transition system is built by the program graph, and the memory has k variables taking values in $\{0, \cdots, n-1\}$, the complexity of the model checking is exponential in the number of variables (n^k).
The complexity depends on the product of the size of the program points, the size of the formula ϕ and n^k.

Model checkers

A model checker is program that can determine whether or not a CTL formula holds on a transition system.

Let $TS = (S, I, \rightarrow, AP, L)$.

Auxiliary functions
Let $S_0 \subseteq S$.

$Reach_1 \quad (S_0) =$
$R := \{\}$
for each $\sigma \rightarrow \sigma'$
if $\sigma \in S_0$ then $R := R \cup \{\sigma'\}$

$Reach \quad (S_0) =$
$R := S_0$
while exists $\sigma \rightarrow \sigma'$ with $\sigma \in R \ \wedge \sigma' \notin R$
R := R $\cup\{\sigma'\}$

Model checkers

To reduce the complexity of model checkers algorithms, we can use the following assumption:
there are transitions leaving all states.

To meet this assumption we add self loops on stuck states.

This allows to have some laws regarding negations.
$AX\phi$ is the same as $\neg EX\neg\phi$

Moreover, we assume that the set of states S is finite.

Let $Sat(\phi)$ be the set of states satisfying ϕ:

$$Sat(\phi) = \{\sigma \mid \sigma \models \phi\}$$

the procedure performs a recursive descent over the formula given as argument

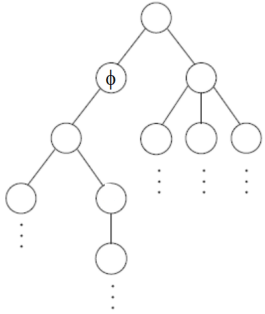
Model checkers

$TS = (S, I, \rightarrow, AP, L)$.

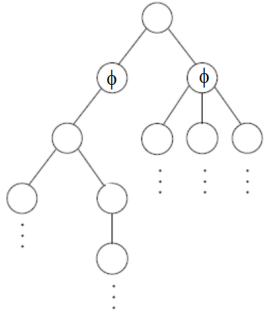
$Sat(tt)$	$=$	S
$Sat(ap)$	$=$	$\{\sigma \in S \mid ap \in L(\sigma)\}$
$Sat(\phi_1 \wedge \phi_2)$	$=$	$Sat(\phi_1) \cap Sat(\phi_2)$
$Sat(\neg\phi)$	$=$	$S / Sat(\phi)$
$Sat(EX\phi)$	$=$	$\{\sigma \mid (Reach_1(\sigma) \cap Sat(\phi)) \neq \{\}\}$
$Sat(AX\phi)$	$=$	$\{\sigma \mid (Reach_1(\sigma) \subseteq Sat(\phi))\}$

Model checkers

$Sat(EX(\phi))$

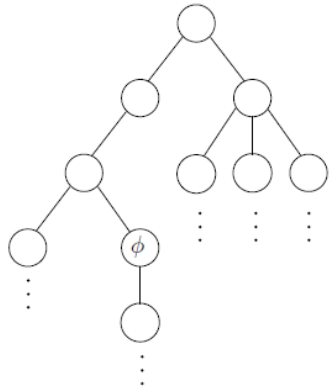


$Sat(AX(\phi))$



Model checkers

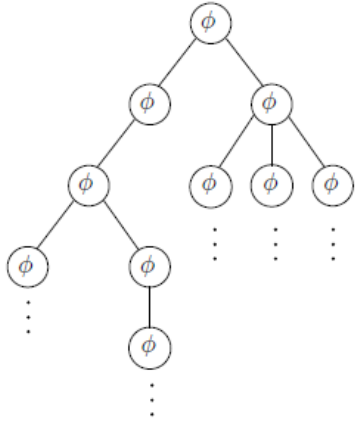
$Sat(EF\phi) = \{\sigma \mid (Reach(\{\sigma\}) \cap Sat(\phi)) \neq \{\}\}$



Another algorithm	
$Sat(EF\phi)$	$=$
	$R := Sat(\phi)$
	while $\sigma \rightarrow \sigma'$ *with* $\sigma \notin R$ *and* $\sigma' \in R$
	do $R := R \cup \{\sigma\}$

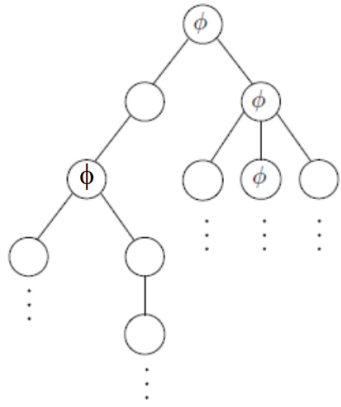
Model checkers

$Sat(AG\phi) = \{\sigma \mid (Reach(\sigma) \subseteq Sat(\phi))\}$



$Sat(AG\phi) = \neg(EF(\neg\phi))$

Model checkers



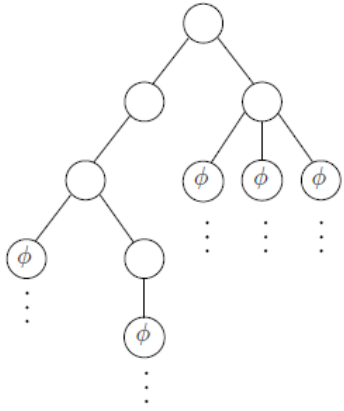
$Sat(EG\phi) = \bigcap_n F^n(Sat(\phi))$
where
$F(S') = \{\sigma \in S' \mid (Reach_1(\{\sigma\}) \cap S') \neq \{\}\}$

$Sat(\phi) = F^0(Sat(\phi))$ and then
we remove states until
each remaining state has a successor
within the resulting set.

Algorithm	
$Sat(EG\phi)$	$=$
	$R := Sat(\phi)$
	while there is $\sigma \in R$ *with* $Rich_1(\sigma) \cap R = \{\}$
	do $R := R/\ \{\sigma\}$

Model checkers

$Sat(AF\phi) = S/ (\bigcap_n F^n(S/ Sat(\phi)))$
where
$F(S') = \{\sigma \in S' \mid (Reach_1(\{\sigma\}) \cap S') \neq \{\}\}$

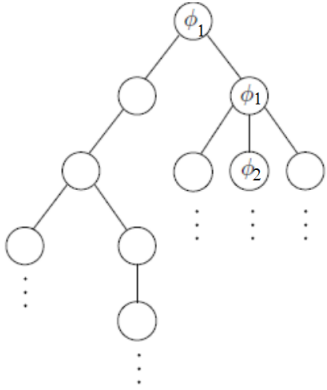


$Sat(AF\phi) = \neg(EG(\neg\phi))$

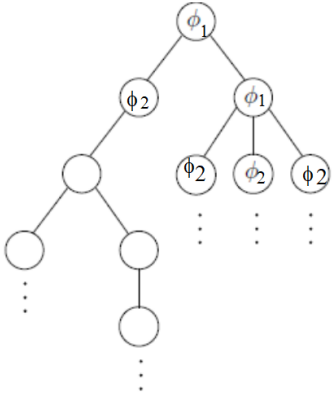
Model checkers

Remaining cases (not shown)

$Sat(E(\phi_1 U \phi_2))$



$Sat(A(\phi_1 U \phi_2))$



from programs to systems

Model checking

Mechanical checking of the satisfaction of a logic formula on the model of the behaviour of the system

Model construction:

- Kripke structure (Transition System - TS)
  $S = \{s_1, \cdots s_n\}$
  $AP = \{p_1, p_2, ...p_k\}$ set of atomic propositions
  $L : S \rightarrow Powerset(AP)\}$

- Labelled transition system (LTS)
  $S = \{s_1, \cdots s_n\}$
  $A = \{a_1, ..., a_m\}$ set of actions
  $\rightarrow$: $S \times A \times S$

LTS can be generated by process algebras specifications.

Process algebras

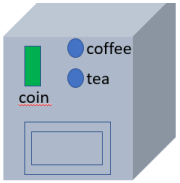
Process algebras are a standard tool for the modelling of concurrent systems.

Assume models given using the Calculus of Communicating Processes (CCS) [Milner, 89].

- A system consists of a set of communicating processes;
- each process executes actions, and synchronizes with other processes.
- Moreover, a special action τ denotes an unobservable action and model internal process actions or internal communications.

Milner, R.: Communication and Concurrency. Prentice-Hall, Inc. (1989)

A tea/coffee vending machine



The behaviour of the machine is the following:

- insert a coin to have a coffee or a tea
- the tea button or the coffee button can be pushed
- after pressing the tea button you collect tea, after pressing the coffee button you collect coffee
- after collecting the tea/coffee, the machine is again available

The machine can described by a recursive process, whose possible actions are:
`coin, teaButton, coffeeButton, collectTea, collectCoffee`.

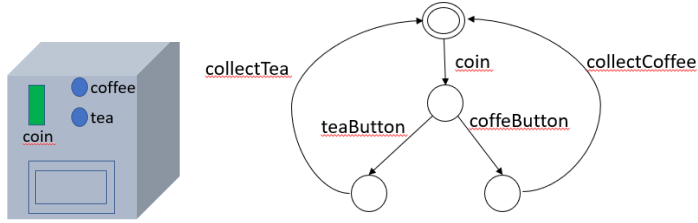
CCS

Syntax and informal semantics of CCS operators.

stop	Inactive process	A process which does nothing
$a : P$	Action prefix	Action a is performed and then process P is executed
$P + Q$	Nondeterministic choice	Alternative choice between the behaviour of process P and that of Q
$P \mid\mid Q$	Parallel Composition	Interleaved execution of process P and process Q
$P \setminus a$	Action restriction	Behaves like P apart from action a that can only be performed within a communication
$P[a/b]$	Action renaming	Behaves like P apart from action a that is renamed b

A tea/coffee vending machine

LTS of M



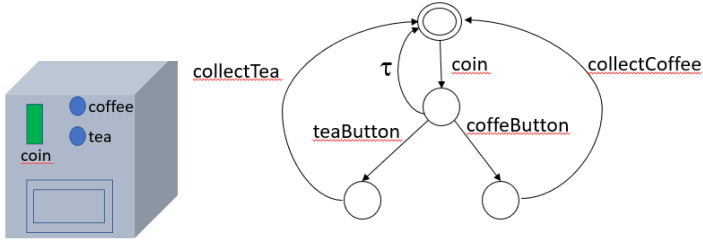
The machine can be described as a recursive process `M`, whose possible actions are: `coin, teaButton, coffeeButton, collectTea, collectCoffee`

```
M = coin :  (P + Q)
P = teaButton :  collectTea :  M
Q = coffeeButton :  collectCoffee :  M
```

A tea/coffee vending machine

Assume the machine can fail: after the user has inserted a coin, the machine not allow to push any button and moves to the initial state, without signalling any failure.

LTS of M




```
M = coin :  (P + Q + R)
P = teaButton :  collectTea :  M
Q = coffeeButton :  collectCoffee :  M
R = tau :  M
```

CCS

- The specification is based on a set *Act* of elementary actions that processes can perform and on a set of operators that permit to build complex processes from simpler ones.
- The special action τ, not belonging to *Act*, represents the unobservable action and is used to model internal process actions or to hide actions to the external environment.
- We denote by $Obs(P)$ the set of observable actions of the process P.

CCS

An inactive process is specified by the **stop** operator.

The action prefix operator specifies the execution of actions in sequence.

The nondeterministic choice operator indicates that a process can choose between the behaviour of several processes.

Parallel composition of two processes corresponds to the interleaved execution of the two processes.

The restriction operator is used to specify processes which synchronise on actions (communication).

A communication transforms the couple of actions executed together into the internal action τ.

The relabeling operator transforms an action into another action.

Labelled Transition Systems

The semantics of process algebras are Labeled Transition Systems (LTSs) which describe the behavior of a process in terms of states, and labeled transitions, which relate states.
An LTS describes sequential nondeterministic behaviours. More formally,

Definition

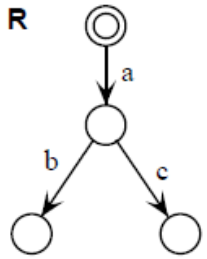
An LTS is a 4-tuple $\mathcal{A} = (X, x^0, Act \cup \{\tau\}, \rightarrow)$, where: X is a finite set of states; x^0 is the initial state; Act is a finite set of observable actions; $\rightarrow \subseteq X \times Act \cup \{\tau\} \times X$ is the transition relation.

We denote by $x \stackrel{a}{\rightarrow} x'$, $a \in Act \cup \{\tau\}$, the transition from the state x to the state x' by executing action a.

Labelled Transition system

Let us consider the process R below. Process R executes action *a* and then may execute action *b* or action *c*, and then stops.

R = a: (R1 + R2)
R1 = b: stop
R2= c: stop

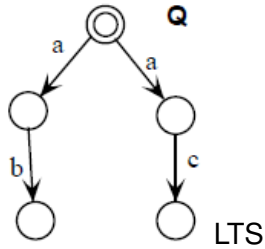


LTS

Labelled Transition System

Let us consider process Q below. Q, differently from R, executes the choice between action *b* and *c* hen performing action *a*.

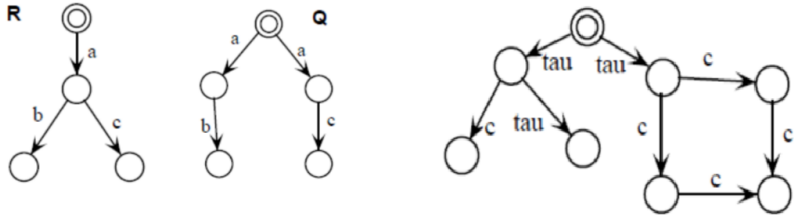
Q = Q1 + Q2
Q1 = a: b: stop
Q2= a: c: stop



LTS

Labelled Transition system

Let us consider the system described by the following process:
$P = (R \parallel Q) \setminus a \setminus b$

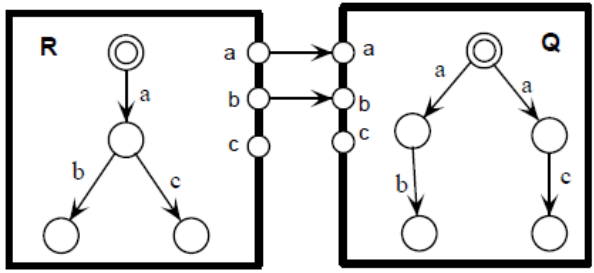


LTS of P

Every state of the LTS represents the combined current states of the subsystems components.
In the initial state, only *a* can be executed. Since *a* is a synchronisation action, *tau* is shown in the LTS.
Then following the left (right) edge of the LTS of *Q*, the behaviour of P is described by the left (right) subtree.

Graphical notation

The graphical specification of process P is the following.



Expressing properties

The temporal logic ACTL (Action-based Computation Tree Logic).

ACTL is an action-based version of the branching time temporal logic CTL.

ACTL has the advantage that, since it is based on actions rather than states, it is naturally interpreted over LTSs.

ACTL

The formulae of ACTL are *action formulae*, *state formulae* and *path formulae*.

An *action formula* permits expressing constraints on the actions that can be observed.

A *state formula* gives a characterization about the possible ways an execution can proceed after a state has been reached.

A *path formula* states properties of an execution.

The truth or falsity of a formula refers to a satisfiability relation over LTSs, denoted $\models$.

Syntax and informal semantics of the used ACTL operators

Action formulae

$\chi ::= \mathit{true}$	any observable action
a	the observable action a
$\sim \chi$	any observable action different from χ
$\chi \mid \chi'$	either χ or χ'

State formulae

$\phi ::= \mathit{true}$	any behaviour is possible
$\sim \phi$	ϕ is impossible
$\phi \,\&\, \phi'$	ϕ and ϕ'
$E\gamma$	there exists an execution in which γ
$A\gamma$	for every execution γ
$< a > \phi$	there exists a next state reachable with a, in which ϕ
$[a]\phi$	for all next states reachable with a, ϕ holds

Path formulae

$\gamma ::= G\phi$	at any time ϕ
$F\phi$	there is a time in which ϕ
$[\phi\{\chi\}U\{\chi'\}\phi']$	at any time χ is performed and also ϕ, *until* χ' is performed and then ϕ'

ACTL formulae

In the table, a is an action belonging to the set *Act* of actions executable by the system, $\sim$ is the *negation* operator, E and A are the existential and universal path quantifiers, while U is the *until* operators.

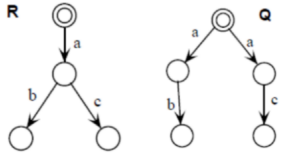
For example, the formula:
$AG([a](< b > true \ \& < c > true))$
states that the system, after having executed the action a, has always the possibility of performing both b and c. This formula is true on the LTS of process R and it is false on the LTS of process Q.

The model checker tool provides a counter-example facility. In the case of satisfied formulae this facility reports a path which verifies the formula; otherwise a path which does not verify the formula is given.

Relation between models

Definition (Traces equivalence)

Traces equivalence considers as equivalent those systems that perform the same sequences of actions.



Q and R have the same traces.

Traces of Q: $\{ab, ac\}$
Traces of R: $\{ab, ac\}$

Relation between models

In general, *system S2 simulates system S1* means that $S2$ observable behaviour is at least as rich as that of $S1$.

The following definition does not cover unobservable behavior, internal computations or hidden communications (τ action).

Definition (Strong simulation)

Let be given a labelled transition system $\mathcal{A} = (Q, q^0, Act, \rightarrow)$.Let $\mathcal{S}$ be a binary relation over Q. Then $\mathcal{S}$ is called a strong simulation over $(Q, Act, \rightarrow)$ if, whenever $p\mathcal{S}q$, if $p \xrightarrow{a} p'$ then there exists $q' \in Q$ such that $q \xrightarrow{a} q'$ and $p'Sq'$.

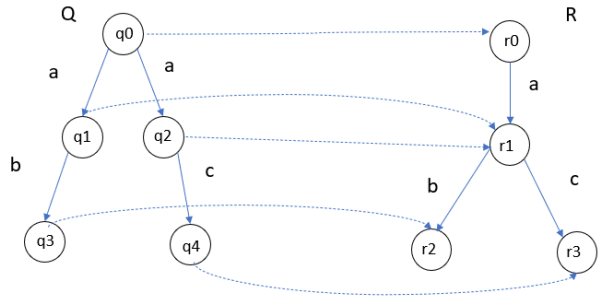
$$\begin{array}{ccc} p & \mathcal{S} & q \\ \downarrow a & & \downarrow a \\ p' & \mathcal{S} & q' \end{array}$$

We say that q strongly simulates p if there exists a strong simulation $\mathcal{S}$ such that $p\mathcal{S}q$.

Strong simulation

The relation between states of a transition system can be easily extended to a relation between two distinct transition systems.

Strong simulation: an example



S: (q0, r0), (q1, r1), (q2, r1), (q3, r2) (q4, r3)

R strongly simulates Q: exists a strong simulation $\mathcal{S}$ such that $q_0 S r_0$.
e.g., $q_0 \xrightarrow{a} q_1$ and there exists r_1 such that $r_0 \xrightarrow{a} r_1$ and $q_1 S r_1$
and $q_0 \xrightarrow{a} q_2$ and there exists r_1 such that $r_0 \xrightarrow{a} r_1$ and $q_2 S r_1$.

R observable behaviour is as reach as that of Q.

The converse $\mathcal{S}^{-1}$ of any binary relation $\mathcal{S}$ is the set of pairs (y, x) such that $(x, y) \in \mathcal{S}$

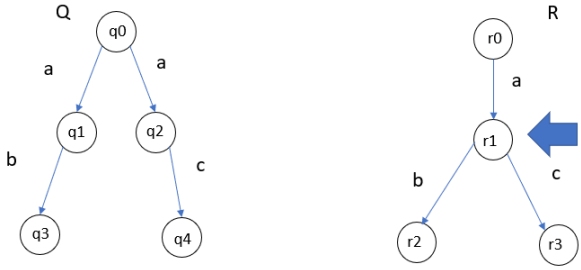
Definition (Strong bisimulation)

Let be given a labelled transition system $\mathcal{A} = (Q, q^0, Act, \rightarrow)$.
Let $\mathcal{S}$ be a binary relation over Q.
Then $\mathcal{S}$ is called a strong bisimulation over $(Q, Act, \rightarrow)$ if both S and its converse $\mathcal{S}^{-1}$ are strong simulations.

There may be several relations that satisfy strong bisimulation.
Let $\sim$ be the maximal strong bisimulation relation.

We say that the states p and q are strongly equivalent, written $p \sim q$.

Strong bisimulation: an example



$\mathcal{S}^{-1}$ is not a strong simulation: Q does not simulate R
State r_1 in R does not have a strongly similar state in Q:
- r_1 is not simulated by q_1, because action c cannot be executed starting by q_1
- r_1 is nor simulated by q_2, because action b cannot be executed starting by q_1

$$Q \not\sim R$$

Strong bisimulation

Definition

Given two processes Q and R, they are strongly bisimilar if and only if a strong bisimulation $\mathcal{S}$ exists which relates the initial states of the LTSs which describe their behavior and we write $Q \sim R$.

Strong Bisimulation

Strong Bisimulation $\sim$

- respects non-determinism. Intuitively, $Q \sim R$ means that Q can do everything that R can do, and vice versa, at every step of the computation
- is an equivalence relation
  - reflexivity: $p \sim p$
  - simmetry: $p \sim q$ implies $q \sim p$
  - transitivity: $p \sim q$ and $q \sim r$ imply $p \sim r$

Moreover

- we can check bisimulation
- there exist algorithms and tools that can generate relations that satisfy the property of being a bisimulation.

Weak bisimulation equivalence

A widely used equivalence is *weak bisimulation*, or *observational* equivalence, which is defined over the set $Act \cup \tau$. The motivation is that only the externally observable actions of a system are relevant in its interaction with the environment.

To abstract unobservable moves during observation, the weak transition relation $\stackrel{a}{\Rightarrow}$ is used.

We have: $\forall a \in Act \stackrel{a}{\Rightarrow} = (\stackrel{\tau}{\rightarrow})^\star \stackrel{a}{\rightarrow} (\stackrel{\tau}{\rightarrow})^\star$, where $\star$ means zero or any number of times.

Two systems are then observationally equivalent whenever no observation can distinguish them.

Weak bisimulation equivalence

Definition (Weak bisimulation)

Let be given a labelled transition system $\mathcal{A} = (Q, q^0, Act, \rightarrow)$.
A weak bisimulation equivalence over Q is a maximal binary bisimulation relation $\mathcal{S}$ such that for every $p, q \in Q$ we have $p\mathcal{S}q$ if and only if: $\forall a \in Act \cup \{\tau\}$:

1. $p \stackrel{a}{\Rightarrow} p', \implies \exists q'$ such that $q \stackrel{a}{\Rightarrow} q'$ and $p'\mathcal{S}q'$.
2. $q \stackrel{a}{\Rightarrow} q', \implies \exists p'$ such that $p \stackrel{a}{\Rightarrow} p'$ and $p'\mathcal{S}q'$.

Definition (observational equivalence)

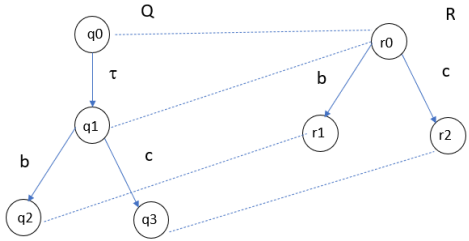
Given two processes Q and QR, they are called *observational equivalent* if and only if a weak bisimulation $\mathcal{S}$ exists which relates the initial states of the LTSs which describe their behavior and we write $Q \approx R$.

Observational equivalence is then defined upon the $\stackrel{a}{\Rightarrow}$ relation.

Observational equivalence

Processes Q and R defined above are not observational equivalent ($Q \not\approx R$), because there exists no state in Q bisimilar to the state in R reached after having executed action a (in this state both action b and c can be performed)

Example of processes which are observational equivalent: $Q \approx R$



$q_0 \stackrel{b}{\Rightarrow} q_2$ $\quad r_0 \stackrel{b}{\Rightarrow} r_1$

$q_0 \stackrel{c}{\Rightarrow} q_3$ $\quad r_0 \stackrel{c}{\Rightarrow} r_2$

S: (q0, r0), (q1, r0), (q2, r1), (q3, r2)

Analysis of models

The LTS describe the behavior of the processes in details, including their internal computations.

- equivalences relations between models
- model checking
  correctness properties are expressed as temporal logic formulae

Assume we use the same formalism to model what is required of a system (its specification) and how it can actually be built (its implementation). Theories based on equivalences can be used to prove that a particular concrete description is correct with respect to a given abstract one.

Similarly, in fault-tolerance and security analysis, the main goal is verifying that a system works correctly in the presence of a given set of *anticipated faults* or attacks.

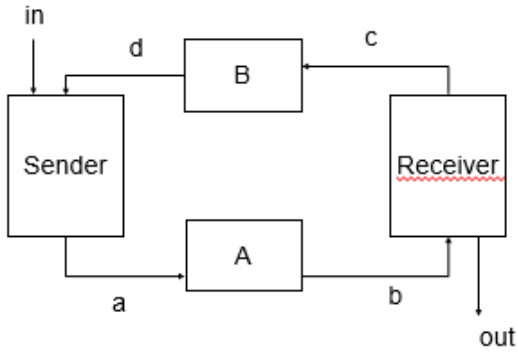
Alternating-bit protocol

The purpose of the protocol is ensuring reliable communication over a medium which may loose messages. A possible implementation of the protocol consists of four processes: the Sender, the Receiver, and two communication channels: one for the delivery of the message, and another for the acknowledgment of message reception.

Sender and Receiver use the value of one bit to identify a message, so that the identifier bit of each message is the complement of the preceding message's bit; a new message is not sent until the sender receives acknowledgment of the current message.

Since the channels can loose messages, both the Sender and the Receiver resend the same message or, respectively, acknowledgment repeatedly until the acknowledgment is received.

Protocol schema



the content of the message is not modeled
alternatig bit
a: a_0, a_1
b : b_0, b_1
c : c_0, c_1
d : d_0, d_1

Actions

Upon an *in* action at the system's external interface, the *Sender* sends the message to the *Receiver* through channel *A*.

Synchronization on action *a*0 or *a*1 depending on the current value of the alternating bit (the first message is identified as 0).

Upon receiving the message, the *Receiver* executes *out*, meaning that the message is available at the interface.

Next, the *Receiver* sends the acknowledgment by synchronizing with channel B on action *c*0 or *c*1 according to the value of the identifier bit of the received message.

Omission of messages or acknowledgments is represented by the τ actions in the processes for the channels, which can take a channel from a state to another without executing the corresponding synchronization action.

For clarity, given an action *a*, we use the notation:
- *a* to denote the sending action
- $\bar{a}$ to denote the complementary receiving action

Moreover we used the character . for the action prefix operator
the notation | for the parallel operator.

The specification is given in the language of the Concurrency Workbench (CWB) model checker.

Alternating-bit protocol

$$P = in.out.P$$

The system is specified by the process *Sys*.

$$Sys = (S_0|A|B|R_1)\backslash L$$

- S_0 is the Sender whose alternating bit is 0;
- R_1 is the Receiver, whose alternating bit is 1;
- A is the delivery channel
- B is the ack channel
- $L = \{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1\}$

Questions

Some questions:

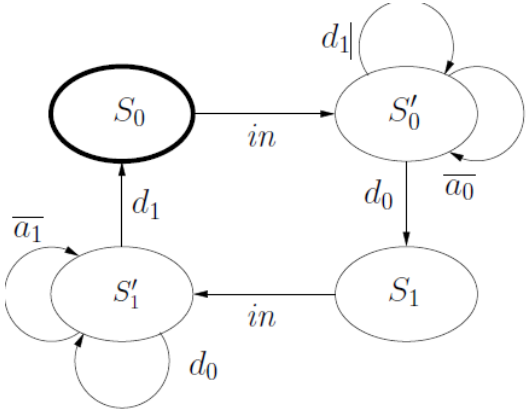
Are P and Sys weak bisimulation equivalent ?

$$P \approx (S_0|A|B|R_1)/L$$

For all paths, is it always possible to execute out?

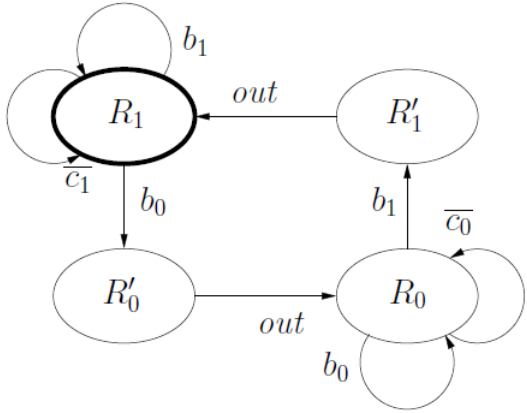
Sender

$$S_0 = in.S'_0$$
$$S'_0 = \overline{a_0}.S'_0 + d_1.S'_0 + d_0.S_1$$
$$S_1 = in.S'_1$$
$$S'_1 = \overline{a_1}.S'_1 + d_0.S'_1 + d_1.S_0$$



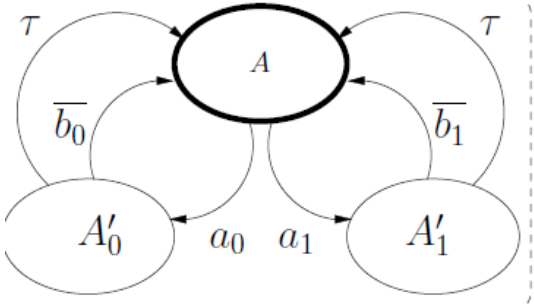
Receiver

$$R_1 = b_0.R'_0 + b_1.R_1 + \overline{c_1}.R_1$$
$$R'_0 = \overline{out}.R_0$$
$$R_0 = b_1.R'_1 + b_0.R_0 + \overline{c_0}.R_0$$
$$R'_1 = \overline{out}.R_1$$



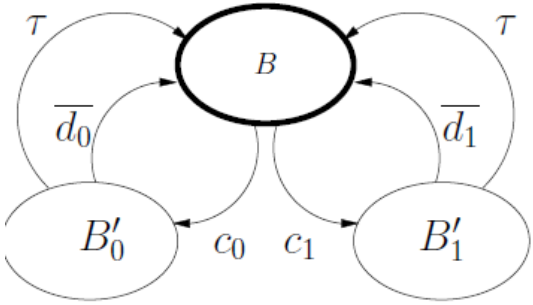
Delivery channel

$$A = a_0.A'_0 + a_1.A'_1$$
$$A'_0 = \overline{b_0}.A + \tau.A$$
$$A'_1 = \overline{b_1}.A + \tau.A$$

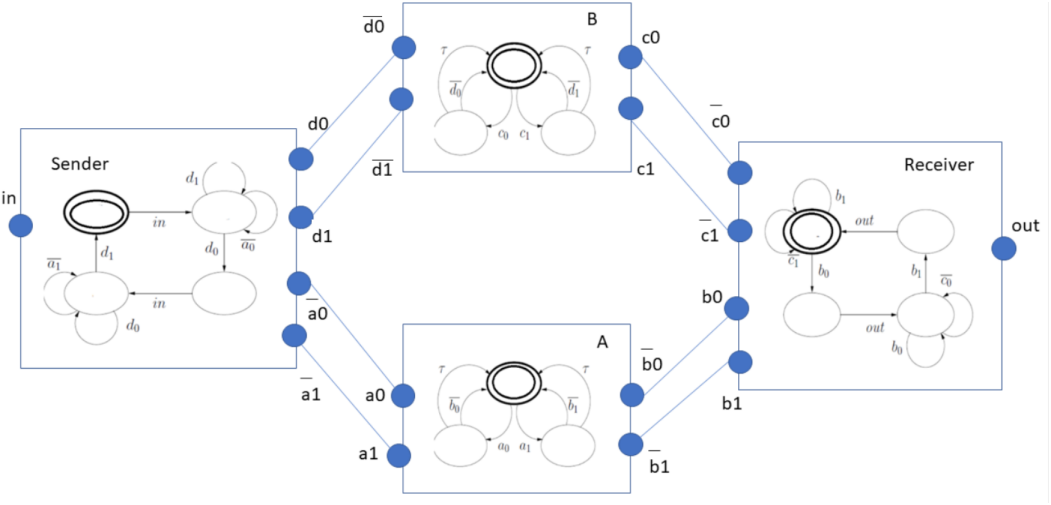


Ack channel

$$B = c_0.B'_0 + c_1.B'_1$$
$$B'_0 = \overline{d_0}.B + \tau.B$$
$$B'_1 = \overline{d_1}.B + \tau.B$$



The system Sys



LTS of the process Sys is generated automatically by the network of processes.

Properties

$$P = in.out.P$$
$$Sys = (S_0|A|B|R_1)\backslash L$$

- P is observational equivalent to Sys: $P \approx Sys$

Using bisimulation, infinite loops of τ could not be detected.

Introduce model checking to complement techniques based on bisimulation.

Properties

- For example, we can express the property that action *out* will eventually be executed.

P is a model for the formula, but the implemenattion, *Sys*, is not a model of the formula.

This is caused by the fact that channels may drop messages or acknowledgments indefinitely by executing τ actions.

Concurrency Workbench of the New Century
Version for MS Windows (.zip): https://sourceforge.net/projects/cwb-nc/
Free Software license.

The logic for expressing properties is $\mu - calculus$.
Examples are available in the archive of CWB-NC.

Matthew Hennessy. Reactive Systems: How to use the Concurrency Workbench; 2008. (https://www.scss.tcd.ie/Matthew.Hennessy/rsexternal/notes/HowTo.pdf)

Running Running CWB-NC

cwb-nc ccs
activates the tool with language CCS

load filename.ccs
reads the input model

eq -S trace P Q
Trace equivalence between P and Q

eq -S bisim P Q
strong bisimulation equivalence between P and Q

eq -S obseq P Q
weak bisimulation equivalence between P and Q

Running CWB-NC

load filename.mu
reads the formulae

chk P fname
checks if P is a model of the formula fname

quit
stops the program

A model checker based on transition systems and CTL

NuSMV: a symbolic model checker
Free Software license.

NuSMV home page: http://nusmv.fbk.eu/

- Modelling the system
- Modelling the properties
- Verification
  - simulation
  - checking of formulae

NuSMV 2.6 documents

NuSMV 2.6 Tutorial.
R. Cavada, A. Cimatti et al., FBK-IRST
Distributed archive of NuSMV (/share/nusmv/doc/tutorial.pdf)

NuSMV 2.6 User Manual.
R. Cavada, A. Cimatti et al., FBK-IRST
Distributed archive of NuSMV (examples/nusmv.pdf)

Examples are available in the archive of NuSMV.

Examples are available also at the URL
<http://nusmv.fbk.eu/examples/examples.html>

Some examples below are taken from the tutorial.

Modelling language

- A system is a program that consists of one or more modules.
- A module consists of
  - a set of state variables;
  - a set of initial states;
  - a transition relation defined over states.
- Every program starts with a module named MAIN
- modules are instantiated as variables in other modules
- Modules can be Synchronous or Asynchronous

Data types

The language provides the following types

- booleans
- enumerations (cannot contain any boolean value (FALSE, TRUE))
- bounded integers
- words: unsigned word[.] and signed word[.] types are used to model vector of bits (booleans) which allow bitwise logical and arithmetic operations (unsigned and signed)
- Arrays
  lower and upper bound for the index, and the type of the elements
  array 0..3 of boolean
  array 10..20 of {OK, y, z}
- .....

Operators

- Logical and Bitwise
  &, |, xor, xnor, ->, <->
- Equality (=) and Inequality (!=)
- Relational Operators >, <, >=, <=
- Arithmetic Operators +, -, * , /
- mod (algebraic remainder of the division)
- Shift Operators «, »
- Index Subscript Operator [ ]
- .....

Other expressions

- **Case expression**
  case
  cond1 : expr1;
  cond2 : expr2;
  ...
  TRUE: exprN;
  esac
- **Next expression**
  refer to the values of variables in the next state
  next(v) refers to that variable v in the next time step
  next((1 + a) + b) is equivalent to (1 + next(a)) + next(b)
  next operator cannot be applied twice, i.e. next(next(a))

Finite state machine-FSM

- Variables
  - state variables
  - input variables
  - frozen variables
    variables that retain their initial value throughout the evolution of the state machine
- transition relation describing how inputs leads from one state to possibly many different states

FMS = finite transition system

Finite Transition system

- Initial state:
  init(<variable>) := <simple_expression> ;
  variables not initialised can assume any value in the domain of the type of the variable

- Transition relation:
  next(<variable>) := <simple_expression> ;
  simple_expression gives the value of the variable in the next state of the transition system

More on variables

- state variables (VAR)
- input variables (IVAR)
  are used to label transitions of the Finite State Machine.
  input variables cannot occur in left-side of assignments
  IVAR i : boolean;
  ASSIGN
  init(i) := TRUE; – legal
  next(i) := FALSE; – illegal
- frozen variables (FROZENVAR)
  variables that retain their initial value throughout the evolution of the state machine
  ASSIGN
  init(a) := d; – legal
  next(a) := d; – illegal

Constraints

- DECLARATION of variables (VAR, IVAR, FROZENVAR)

- ASSIGNMENTS that define the inital states

- ASSIGNMENTS that define the transition relation

Assignments describe a system of equations that say how the FSM evolves through time.

ASSIGN a := exp;
ASSIGN init(a) := exp
ASSIGN next(a) := exp

Constraints

DEFINE is used for abbreviations
DEFINE <id> := <simple_expression> ;
no constraint on order where a declaration of a variable should be placed

FAIRNESS constraint
A fairness constraint restricts to fair execution paths. Paths that satisfy the expression simple_expr below, which is assumed to be boolean.
When evaluating formulae, the model checker considers path quantifiers to apply only to fair paths.

FAIRNESS simple_expr ;

Module declaration

A module declaration is a collection of declarations, constraints and specifications (logic formulae).

A module can be reused as many times as necessary. Modules are used in such a way that each instance of a module refers to different data structures.

A module can contain instances of other modules, allowing a structural hierarchy to be built.

module :: MODULE identifier [(module_parameters)] [module_body]

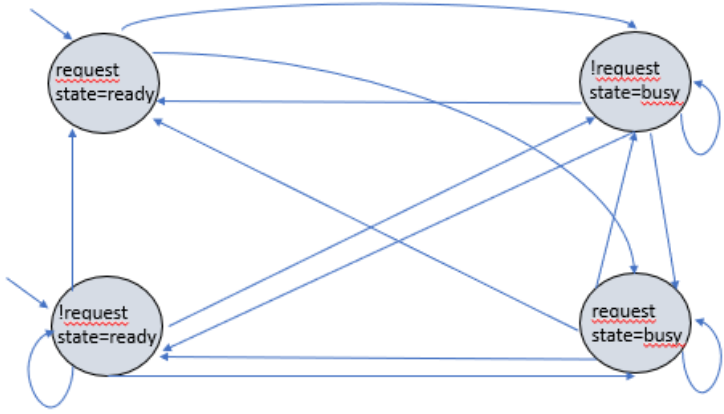
A simple program

A system can be ready or busy. Variable state is initially set to ready. Variable request is an external uncontrollable signal. When request is TRUE and variable state is ready, variable state becomes busy. In any other case, the next value of variable state can be ready or busy: request is an unconstrained input to the system.

```
MODULE main
VAR
request : boolean;
state: {ready, busy };
ASSIGN
init(state) := ready;
next(state) := case
        state = ready & request = TRUE : busy;
        TRUE: {ready, busy };
        esac;
```

A simple program

Build the transition system (also named Finite state machine - FSM)
4 states
2 initial states
14 transitions



Running NuSMV

./NuSMV -int
activates an interactive shell for simulation

read_model [-i filename]
reads the input model

go
reads and initializes NuSMV for simulation

reset
resets the whole system

help
shows the list of all commands

quit
stops the program

Simulation

pick_state [-v] [-r | -i]
picks a state from the set of initial states
-v prints the chosen state.
-r pick randomly
-i pick interactively

simulate [-p | -v] [-r | -i] -k
generates a sequence of at most k steps
starting from the current state
-p prints only the changed state variables
-v prints all the state variables
-r at every step picks the next state randomly
-i at every step picks the next state interactively

Simulation

goto state state label
makes state label the current state (it is used to navigate along traces).

show_traces [-v] [trace number]
shows the trace identified by trace number or the most recently generated trace. -v prints prints all the state variables.

print_current_state [-v]
prints out the current state.
-v prints all the variables

An interactive session

```
./NuSMV -int
read_model -i file.smv
go
pick_state -r
print_current_state -v
simulate -v -r -k 3
show_traces -t
show_traces -v
```

___

pick with constraint

```
pick_state -c "request = TRUE" -i
```

Verification

Specifications written in CTL can be checked on the FSM .

OPERATORS:
EX p
AX p
EF p
AF p
EG p
AG p
E[p U q]
A[p U q]

A CTL formula is true if it is true in all initial states.

Checking properties

1. Specify the formula:

MODULE ...
.....

SPEC ... CTL formula ....

2. Invoke NuSMV as follows:

./NuSMV file.smv

An example

(– is a commented line in the file .smv)

```
MODULE main
VAR
request : boolean;
state: {ready, busy };
ASSIGN
init(state) := ready;
next(state) := case
        state = ready & request = TRUE : busy;
        TRUE: {ready, busy };
        esac;

SPEC AG (state = busy | state= ready);
SPEC EF (state = busy);
SPEC EG (state = busy);
– SPEC AG (state=ready & request=true) -> AX state = busy;
```

A system with more than one module

MODULE instantiation

An instance of a module is created using the VAR declaration. In the declaration actual parameters are specified

In the following example, the semantic of module instantiation is similar to call-by-reference (the variable a below is assigned the value TRUE)

```
MODULE main
VAR
a : boolean;
b : foo(a);
...

MODULE foo(x)
ASSIGN
x := TRUE;
```

MODULE instantiation

In the following example, the semantic of module instantiation is similar to call-by-value

```
MODULE main
...
DEFINE
a := 0;
VAR
b : bar(a);          b is a module of type bar declared inside module main
...

MODULE bar(x)
DEFINE
a := 1;
y := x;
```

The value of y is 0

Composition of modules

```
MODULE mod
VAR
out: 0..9;
ASSIGN
next(out) := (out + 1) mod 10;

MODULE main
VAR
m1 : mod;
m2 : mod;
sum: 0..18;
ASSIGN sum := m1.out + m2.out;
```

. used to access the components of modules (e.g., variables)
self used for the current module

Composition of modules

Module declarations may be parametric.

```
MODULE mod(in)
VAR out: 0..9;
...

MODULE main
VAR
m1: mod(m2.out);
m2 : mod(m1.out);
...
```

Composition of modules

- modules have parameters (input/output parameters)
- variables declared in a module are local to the module
- synchronous composition: all modules move at each step (by default)
- aynchronous composition (modules instantiated with the keyword **process**): one process moves at each step (it is possible to define a collection of parallel processes, whose actions are interleaved, following an asynchronous model of concurrency)

Processes

One process is non-deterministically chosen, and the assignment statements declared in that process are executed in parallel. Variables not assigned by the process remains unchanged. Next process to execute is chosen non-deterministically.

running: a special variable of each process - TRUE if and only if that process is currently executing. It can be used in a fairness constraint (formula true infinitely often).

Exercise: A synchronous three bit counter.

```
MODULE main
VAR
bit0 : counter_cell(TRUE);
bit1 : counter_cell(bit0.carry_out);
bit2 : counter_cell(bit1.carry_out);

MODULE
counter_cell(carry_in)
VAR
value : boolean;
ASSIGN
init(value) := FALSE;
next(value) := value xor carry_in;
DEFINE
carry_out := value & carry_in;

SPEC AG AF bit2.carry_out
```

Exercise: A mutual exclusion problem

Implement mutual exclusion between two processes, using a boolean variable semaphore.

Each process has four states: idle, entering, critical and exiting.

The entering state indicates that the process wants to enter its critical region.

If the variable semaphore is FALSE, it goes to the critical state, and sets semaphore to TRUE.

On exiting its critical region, the process sets semaphore to FALSE again.

Exercise

```
MODULE main
VAR
semaphore : boolean;
proc1: process user(semaphore);
proc2: process user(semaphore);

ASSIGN
init(semaphore) := FALSE;

MODULE user(semaphore)
VAR
state : {idle, entering, critical, exiting};
.....
```

```
MODULE user(semaphore)
VAR state : {idle, entering, critical, exiting};

ASSIGN
init(state) := idle;
next(state) := case
    state = idle : {idle, entering}
    state = entering & !semaphore : critical
    state = critical : {critical, exiting}
    state = exiting : idle
    TRUE : state
    esac;
next(semaphore) := case
    state = entering : TRUE
    state = exiting : FALSE
    TRUE : semaphore
    esac;
FAIRNESS
running
```

Exercise

Properties

1. It never is the case that the two processes proc1 and proc2 are at the same time in the critical state

AG ! (proc1.state = critical & proc2.state = critical)

2. if proc1 wants to enter its critical state, it eventually does - a liveness property

AG (proc1.state = entering -> AF proc1.state = critical)

Counter-example path. It can happen that proc1 never enters its critical region.

Another way to model a system

- INIT constraint
  The set of initial states of the model is determined by a boolean expression under the INIT keyword.
- INVAR constraint
  The set of invariant states can be specified using a boolean expression under the INVAR key- word.
- TRANS constraint
  The transition relation of the model is a set of current state/next state pairs. Whether or not a given pair is in this set is determined by a boolean expression, introduced by the TRANS keyword.