Model checking

C.Bernardeschi

Model checking

A fully automated method for verifying properties of programs/systems.
Closer to program verification than to program analysis.

Model checking technique:

- the description of the program/system is given by a transition system, typically an abstraction of the concrete system or program.
- the property is specified by a formula of a temporal logic
- the method of examination is given by an algorithm for checking the truth of the logic formula in the given transition system (i.e, checking whether the state transition system is a model of the logic 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.

Temporal logic

- *Temporal logic* is a logic for representing and reasoning about propositions qualified in terms of time. For example, "the traffic-light is always red" or "the traffic-light will eventually be red".

- In the context of model checking, two notions of time and corresponding temporal logics are commonly used:

  Branching time: different possible future states are considered (CTL – Computation Tree Logic).

  $$\begin{array}{ccc} & s_0 & \\ \swarrow & & \searrow \\ s_1 & & s_2 \\ \cdots & & \cdots \end{array}$$

  Linear time: A linear time line, represented by a totally order sequence of points in time is considered (LTL – Linear-time Temporal Logic).

  $$s_0 \rightarrow s_1 \rightarrow s_2 \cdots$$

We consider branching time logic.

Transition System

We can assume a set of *atomic propositions* and states are assigned a subset of such 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;
- $L : \sigma \Rightarrow PowerSet(AP)$ is a labelling function for states.
  We write
  $\sigma \rightarrow \sigma'$ if there is a transition from state σ to state σ'.

Computation Tree Logic - CTL

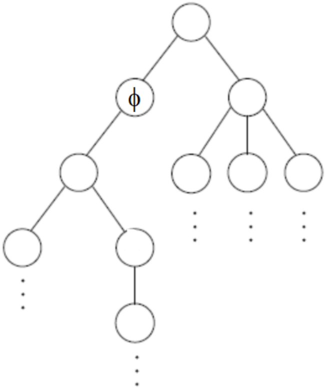
Atomic proposition (the simplest formula)

ϕ atomic proposition

Example:
light-green, light-red, power-on, power-off, ...

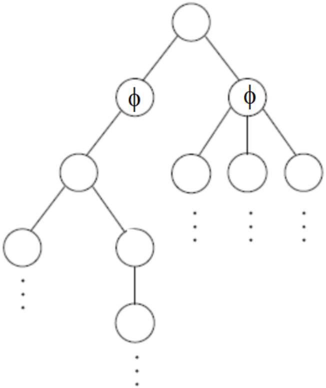
Reachability in one step

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



Reachability in one step

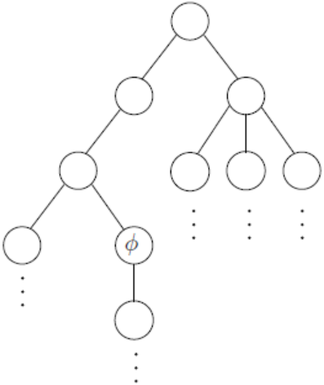
$AX\phi$ the next state it is certain that satifies ϕ



Reachability

EF ϕ

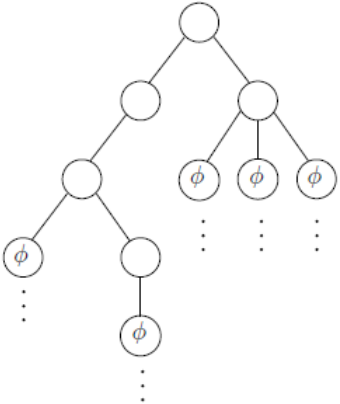
There exists a path and a state along that path such that ϕ holds



Reachability

AF ϕ

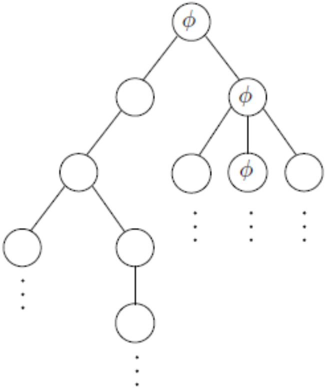
Along every path there exists a state such that ϕ holds at that state



Unavoidability

EG ϕ

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

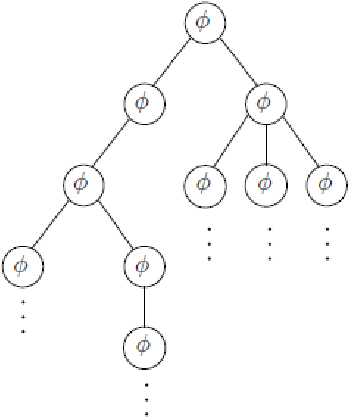


CTL

Unavoidability

AG ϕ

Along every path ϕ holds at every state



CTL

The semantics of CTL formulas is defined by reference to state transition systems as models.

The behaviour of a system is described by the possible execution paths, i.e. sequence of transitions.

A formula ϕ holding in a state σ of model is expressed by:

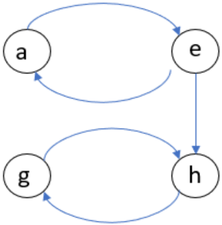
$$\sigma \models \phi$$

An example

$S = \{a, e, g, h\} \quad I = \{a\} \quad AP = \{consonant, vowel\}$

$L(a) = \{vowel\}, L(e) = \{vowel\}$

$L(g) = \{consonant\}, L(h) = \{consonant\}$



Atomic proposition

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

Reachability in one step

$e \models EX\ consonant \qquad a \not\models EX\ consonant$

$a \models AX\ vowel \qquad e \not\models AX\ vowel$

Reachability

$a \models EF\ consonant \qquad g \not\models EF\ vowel$

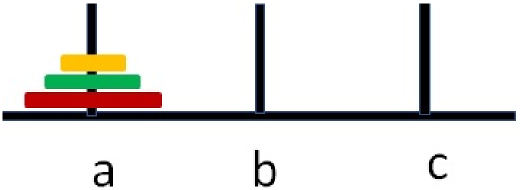
$a \models AF\ vowel \qquad e \not\models AF\ vowel$

Unavoidability

$a \models EG\ vowel \qquad g \not\models EG\ vowel$

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

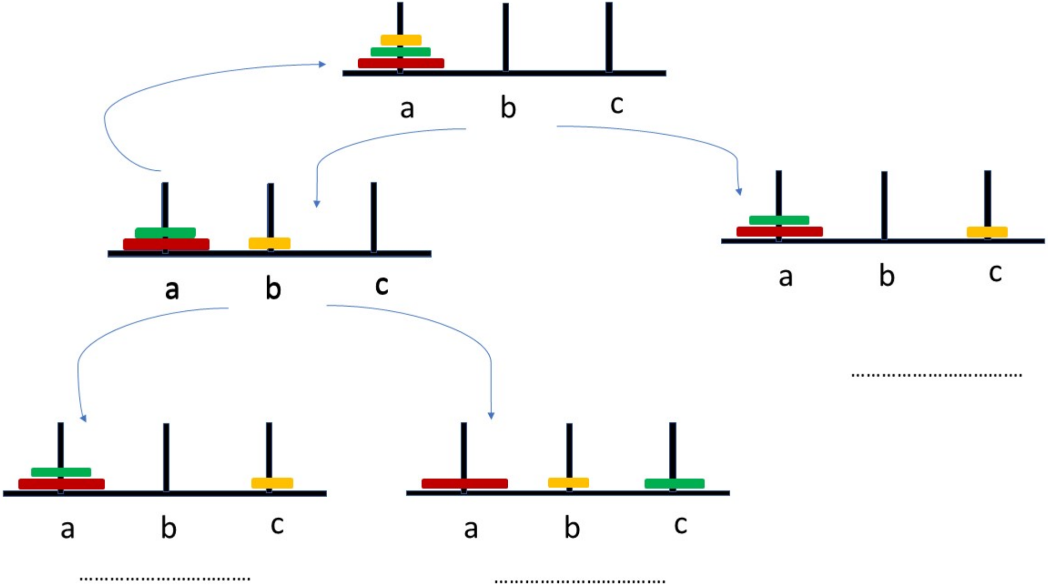
Another example of Transition System
Towers of Hanoi



- 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

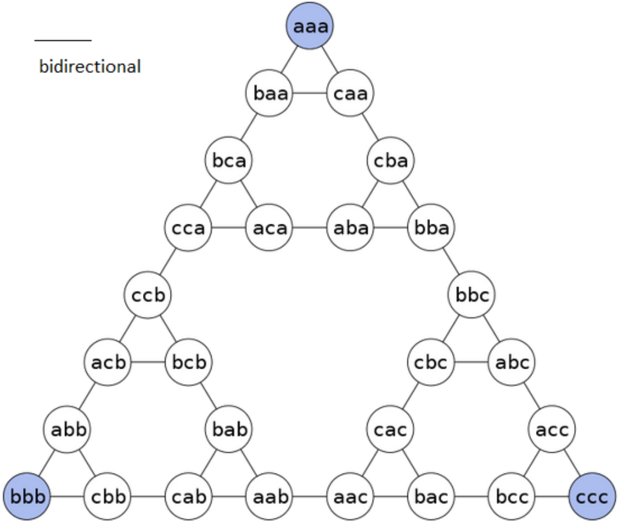
https://en.wikipedia.org/wiki/Tower_of_Hanoi

Hanoi Tower: steps



Transition System

State: sequence of rods on which disks are placed (pos_{small} pos_{medium} pos_{large}).
State `aaa` represents the state in which all disks are placed on rod *a*.



Transition System

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

- S is the set of sequences of three letters chosen among a, b, c;
- $I = \{\mathtt{aaa}\}$;
- $\rightarrow$ 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 $\mathtt{c}$".

Let $\phi = ccc$

$$\text{aaa} \models AF\ \phi$$

Computation Tree Logic

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

The relation $\models$ is defined inductively over the structure of CTL formulas.
Let $\mathit{Path}(\sigma)$ be the set of paths starting at σ.

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

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, $X\phi$ holds on a path, whenever ϕ holds in some successor state
$F\phi$ holds on a path, whenever ϕ holds in some state of the path, possibly the current state.

Definitions

- a state formula ϕ holds on a transition system 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(σ_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*$(\sigma) = \sigma$. This condition ensures that all paths are infinite.

CTL formulas

Certain forms of formulas occur frequently in specifications of practically relevant properties.

Examples

For any state, when a resource is requested it will also be ready eventually:

$$AG\,(\mathit{requested} \implies AF\ \mathit{ready})$$

Resource ready is true infinitely often on every execution path:

$$AG\,(AF\ \mathit{ready})$$

Resource released will be reached in any case:

$$AF\,(AG\ \mathit{released})$$

It is always (i.e. from every state) possible to reach the start state:

$$AG\,(EF\ \mathit{start})$$

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

- $Reach(I)$
  the complete set of reachable states

Analysis of programs

Atomic propositions in the analysis of programs

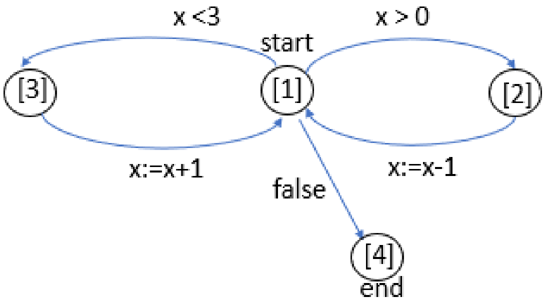
Program = Program Graph + Data

- 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 $(x, 3)$.

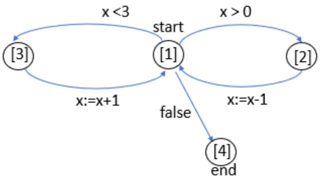
Transition System

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

Atomic propositions
$\{@[i], i = 1, 2, 3, 4\} \cup \{@(x, i), i = 0, 1, 2, 3\} \cup \{start\} \cup \{end\}$

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 Relation



Definition of the transition relation

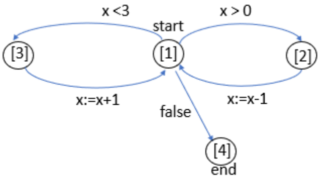
Assume:
$n \in \{0, 1, 2, 3\}$ is the value of x
$q \in \{a, b, c, d\}$ is the value of the program point,
with $a = [1], b = [2], c = [3], d = [4]$

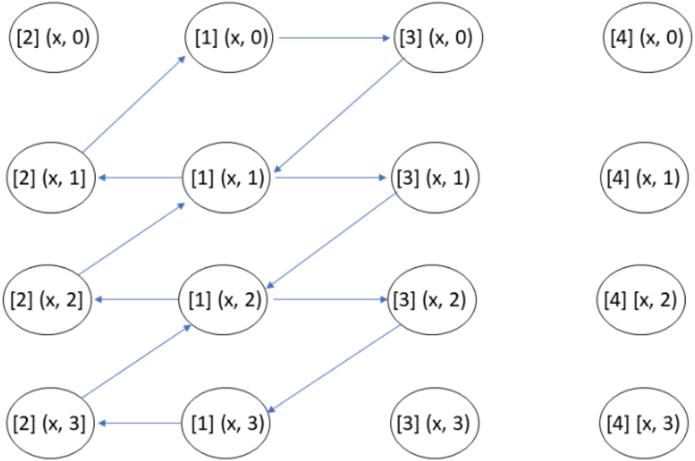
$qn \rightarrow qn'$ iff:

$$
\begin{aligned}
&(n = n' \land q = a \land q' \in \{b, c\} \land \\
&\quad (q' = b \implies n < 0) \land (q' = c \implies n < 3)) \lor \\
&(n' = n + 1 \land q = c \land q' = a) \lor \\
&(n' = n - 1 \land q = b \land q' = a)
\end{aligned}
$$

Transition System TS

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





Analysis of programs

How many states? How many reachable states?

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

Property: for each initial state it is possible to terminate

$$start \implies EF\ end$$

Property: for each initial state it is certain to terminate

$$start \implies AF\ end$$

Analysis of programs

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 $S_0 \subseteq S$.

$$
\begin{array}{ll}
\mathit{Reach}_1 & (S_0) = \\
 & R := \{\} \\
 & \textit{for each } \sigma \rightarrow \sigma' \\
 & \quad \text{if } \sigma \in S_0 \quad \text{then} \quad R := R \cup \{\sigma'\}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Reach} & (S_0) = \\
 & R := S_0 \\
 & \textit{while exists } \sigma \rightarrow \sigma' \quad \text{with} \quad \sigma \in R \ \wedge \sigma' \notin R \\
 & \quad \text{R := R} \cup \{\sigma'\}
\end{array}
$$

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

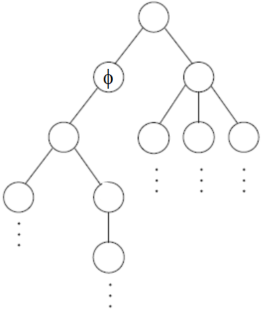
Model checkers

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

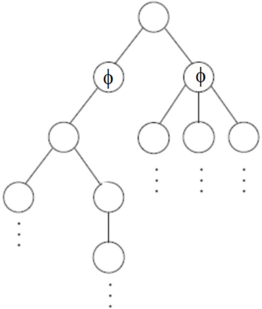
$$
\begin{array}{lcl}
\hline
Sat(tt) & = & S \\
Sat(ap) & = & \{\sigma \in S \mid ap \in L(\sigma)\} \\
Sat(\phi_1 \wedge \phi_2) & = & Sat(\phi_1) \cap Sat(\phi_2) \\
Sat(\neg\phi) & = & S /\ Sat(\phi) \\
Sat(EX\phi) & = & \{\sigma \mid (Reach_1(\sigma) \cap Sat(\phi)) \neq \{\}\} \\
Sat(AX\phi) & =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

Model checkers

$Sat(EX(\phi))$

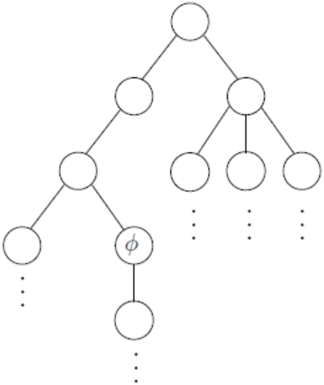


$Sat(AX(\phi))$



Model checkers

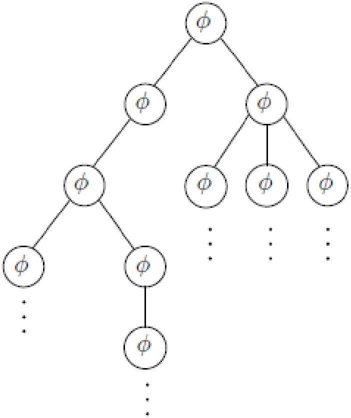
$$Sat(EF\phi) \;=\; \{\sigma \mid (Reach(\{\sigma\}) \cap Sat(\phi)) \neq \{\}\}$$



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

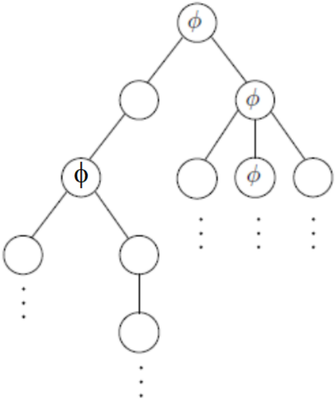
Model checkers

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



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

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.

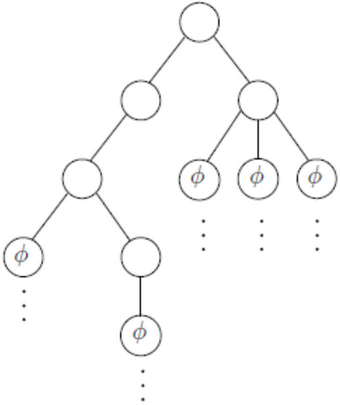
Algorithm	
$Sat(EG\phi)$	$=$
	$R := Sat(\phi)$
	while there is $\sigma \in R$ *with* $Rich_1(\sigma) \cap R = \{\}$
	do $R := R/\ \{\sigma\}$

Model checkers

$$Sat(AF\phi) \;=\; S/\,(\bigcap_n F^n(S/\,Sat(\phi)))$$

where

$$F(S') = \{\sigma \in S' \mid (Reach_1(\{\sigma\}) \cap S') \neq \{\}\}$$



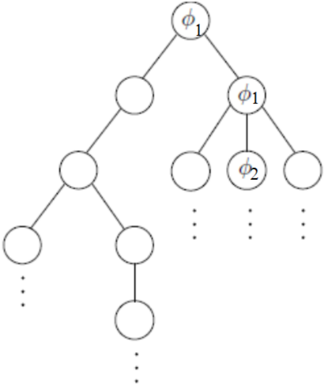
$$Sat(AF\phi) \;=\; \neg(EG(\neg\phi))$$

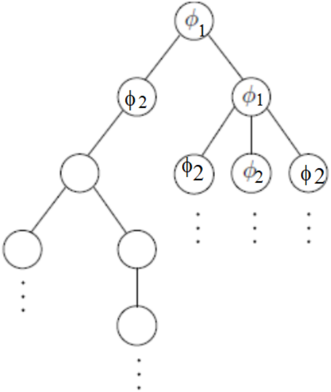
Model checkers

Remaining cases (not shown)

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

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





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;
```

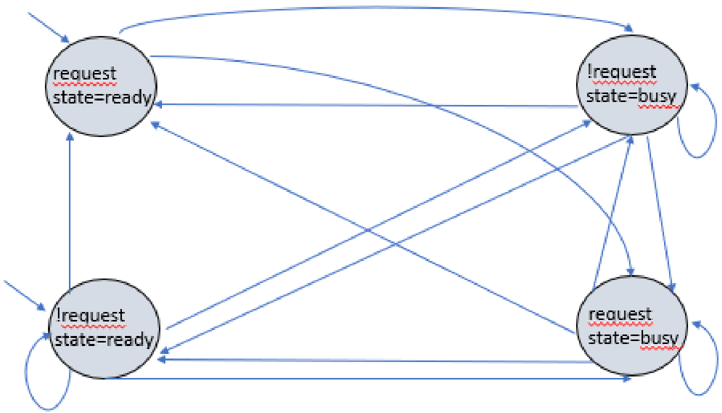
A simple program

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.