Symbolic Model Checking

- Part I -
LTL Model Checking - NuSMV

Gabriel Vilallonga
UNICAM – UNCa
May 2012
Simbolic Model Checking

- Part I -

- Introduction to Model Checking
- LTL
- Introduction to NuSMV
Symbolic Model Checking

• Partial contents come from Edmund Clark slides.
• I also utilized A. Cimatti and M. Pistore material.
Simbolic Model Checking - Intro

- A practical tutorial on Model Checking
- Course Mission
- A practical introduction to symbolic model checking,
- The use of NuSMV symbolic model checker.
What is a Model Checker?

```
G(p -> Fq)
```

```
finite-state model
```

```
Model Checker
```

```
temporal formula
```

YES!

```
q
```

```
p
```

NO!

```
counterexample
```

Gabriel Vilallonga – UNICAM -UNCa
Simbolic Model Checking - Intro

- Model Checking is a formal verification technique
- From academics to industry in a decade
- Does not require deep training
- Powerful debugging capabilities
Where does Model Checking come from?

Concurrent Program Verification

- Early Stages: Hand constructed proofs

- Late '70, early '80
  Temporal Logic to express:
  - mutual exclusion,
  - absence of deadlock, and
  - absence of starvation
SMC - Introduction

Early '80 Emerson & Clarke
State-Exploration with Temporal Logic

"The proof construction can be replaced by a model theoretic approach which will mechanically determine if the system meets a specification expressed in proportional temporal logic"
Advantages

• Doesn't need to construct a correctness proof.

• Faster than other rigorous methods.

• The MC produce a counterexample execution trace.

• No problems with partial specifications.

• Temporal Logic can express many of the concurrent systems properties.
Disadvantages

The objections:

• Proving a program helps you understand it
  “Suffering makes us stronger“

• Temporal logic specifications are ugly...

• Writing specifications is hard

• State explosion is a major problem.... absolutely true.
A Kripke model for mutual exclusion

N = noncritical, T = trying, C = critical

User 1  User 2
SMC – Intro – Kripke Structure

- Graph having the reachable states of the system as nodes and system state transitions as edges
- Contains a labeling of the states of the system with properties that hold in each state
- Needs to fix a set of atomic propositions AP, which denote the properties of individual states
Kripke models are used to describe reactive systems:

- nonterminating systems with infinite behaviors,
- represent dynamic evolution of modeled systems;
- values to state variables, program counters, content of communication channels.

Formally, a Kripke model \((S, R, I, L)\) consists of

- a set of states \(S\);
- a set of initial states \(I \subseteq S\);
- a set of transitions \(R \subseteq S \times S\);
- a labeling \(L \subseteq S \times AP\)

\(AP\) is a non-empty set of Atomic Propositions.
Let $M = (S; s_0; R; L)$, a Kripke structure, a finite path that is a non-empty finite sequence of states $\pi = \langle \pi_0, \ldots, \pi_{n-1} \rangle$, $\pi_0 = s_0$, $\pi_0, \ldots, \pi_{n-1}$ $S$ such that $(\pi_i, \pi_{i+1}) \in R$, $i, 0 \leq i \leq n-1$.

$|\pi| = n$ is the path length.

An infinite path is an infinite sequence of states

$\langle \pi_0, \pi_1, \pi_2, \ldots \rangle$, $(\pi_i, \pi_{i+1}) \in R$, $\pi_i \in S$, $0 \leq i$.

The path length is $\infty$.  

∀i, 0 ≤ i ≤ |π|, \( n_i \) denotes the i\(^{th}\) state in the path

Let \( \langle \pi_0, \ldots, \pi_{n-1} \rangle \),

\( n^i \) represent the sequence \( \langle \pi_i, \pi_{i+1}, \ldots \rangle \), the path queue starting from \( n_i \).
Temporal logics describe the ordering of events in time without introducing time explicitly.

Classified according to the time structure:
  - linear, or
  - branching

The meaning of a temporal logic formula is determined with respect to a labeled state-transition graph or Kripke structure.
Linear Time Temporal Logic (LTL)
   – interpreted over each path of the Kripke structure
   – linear model of time
   – temporal operators

Properties such as “for some state on the path” or “for every two consecutive states” can be expressed.

Propositional logic
   – $a \in AP$ atomic proposition
   – $\neg \phi$ and $\phi \land \psi$ negation and conjunction
SMC – Linear Temporal Logic

Temporal operator

\[ X(\phi) \quad \text{neXt} \]
\[ U(\phi, \psi) \quad \text{Until} \]
\[ F(\phi) \quad \text{Finally} \]
\[ G(\phi) \quad \text{Globally} \]
An LTL formula is interpreted over a Kripke structure path
\[ \pi = \langle \pi_0, \ldots, \pi_\infty \rangle \]

When a Kripke structure satisfies a LTL formula \( \phi \) is expressed by (\( \pi \models \phi \))

\[ \pi \models p \text{ iff } p \in L(\pi_0). \]

\( \pi \) Satisfies an AP \( p \) if the first state of the sequence hold \( p \).

\[ \pi \models \neg \phi \text{ iff } \pi \not\models \phi \]
\[ \pi \models \phi_1 \lor \phi_2 \iff \pi \models \phi_1 \circ \pi \models \phi_2 \]

\neg y \lor \text{the typical usage in the first order logic.}

We can work with the usual equivalences such as

\[ \phi_1 \land \phi_2 \equiv \neg (\neg \phi_1 \lor \neg \phi_2) \]
SMC – Linear Temporal Logic

Next Operator

\[ \pi \models X(\phi) \text{ iff } |\pi| > 1 \text{ y } \pi_1 \models \phi \]

\[ \langle \pi_0, \pi_1, \pi_2, \ldots \rangle \]

The formula \( X(\phi) \) means that \( \phi \) is true at the next state, \( \pi_1 \), after the initial one, that is, at \( \pi_0 \)
The formula holds $U(\phi, \psi)$, if $\phi$ holds until $\psi$ occurs, i.e., there is a state on the path at which $\psi$ holds, and at every state before $\phi$ holds.

\[ \langle \pi_0, \ldots, \pi_{k-1}, \pi_k, \ldots \pi_\infty \rangle \]
Finally Operator

\[ \pi \models F(\phi) \text{ iff exist } k, 0 \leq k \leq |\pi|, \quad \pi_k \models \phi \]

The formula \( F(\phi) \) holds, if \( \phi \) eventually occurs, i.e., \( \phi \) holds at some state on the path.

\[ \langle \pi_0, \ldots, \pi_K, \ldots \rangle \]

\[ F(\phi): \quad \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \phi \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \cdots \]
Globally Operator

\[ \pi \models G(\phi) \text{ iff for all } k, \ 0 \leq k \leq |\pi|, \text{ where } \pi_k \models \phi \]

The formula \( G(\phi) \) holds, if \( G(\phi) \) holds globally, i.e., at every state along the path.

\[ \langle \pi_0, \ldots \pi_\infty \rangle \]

\[ G(\phi): \quad \phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi \ldots \]
SMC – Linear Temporal Logic

Strong Until y Weak Until

\[ U(\phi, \psi) = WU(\phi, \psi) \land F(\psi) \quad \text{and} \quad WU(\phi, \psi) = U(\phi, \psi) \lor G(\phi) \]

Duales

\[ \neg U(\phi, \psi) = WU(\neg \psi, \neg \phi \land \neg \psi) \quad \text{and} \quad \neg WU(\phi, \psi) = U(\neg \psi, \neg \phi \land \neg \psi) \]
• F may be define in term of U,
• G en term of de WU

\[ F(\phi) = U(\text{true}, \phi) \quad \text{and} \quad G(\phi) = WU(\phi, \text{false}) , \]

• F and G are duales

\[ F(\phi) = \neg G(\neg \phi) \quad \text{and} \quad G(\phi) = \neg F(\neg \phi) . \]
SMC – Linear Temporal Logic

state formula
\( p \in \text{Ap} \)

path formula
\( \neg p, p \land q, p \lor q \),
\( G f, F f, Xf \),
\( f U g, f R g \)

LTL formula
\( Af \)
(with \( f, g \) path formulas)
(with \( f \) path formula)
Properties of Transition Systems in LTL

The LTL formulas express properties of paths in computation trees.

Reachability and safety properties.

A state is called *reachable* if there is a computation path from an initial state leading to this state.
Reachability is one of the most important properties of transition systems in connection with safety properties.

The system is safe if one cannot reach a state at which unsafe holds.

F unsafe

Then safety of the system can be expressed as non-reachability of a state satisfying unsafe, i.e., the property

G ¬ unsafe
Given a vending machine

An unsafe behavior would be serving beer to a professor, which can be expressed by the formula

\[ \text{disp} = \text{beer} \land \text{customer} = \text{prof} \]

Thus, the corresponding safety property is:

\[ G (\text{disp} \neq \text{beer} \lor \text{customer} \neq \text{prof} ) \]

Another safety: there is always either coffee or beer in the storage. It can be expressed by the following formula:

\[ G (\text{st _ cofee} \lor \text{st _ beer}) \]
Mutual exclusion
Property of concurrent systems.

*It arises when two or more processes are not allowed to enter the same critical section of a concurrent system simultaneously.*

Assuming two processes $P_1; P_2$, and that formulas $\text{critical}_i$, where $i = 1; 2$ denote that $P_i$ is in the critical section,

The mutual exclusion formula is:

$$G \neg(\text{critical}_1 \land \text{critical}_2)$$
For the vending machine example:

“Coffee and beer cannot be in the dispenser simultaneously”

\[ \neg (\text{disp} = \text{coffee} \land \text{disp} = \text{beer}) \]

We may want to express that a professor and a student cannot be customers at the same time:

\[ \neg (\text{customer} = \text{student} \land \text{customer} = \text{prof}) \]
Deadlock.
A concurrent program is in a *deadlock* situation, when no terminal state is reached, and no part of the program is able to proceed.

A transition system or Kripke structure is said to be deadlock-free if no computation in it leads to a deadlock.

Assuming that the set of terminal states is represented by a temporal formula terminal, we can express deadlock-freedom by the formula

$$G(N \perp \rightarrow \text{terminal})$$
This formula must be true on every path. Indeed, it is easy to see that the formula $N \perp$ means “there is no next state”, that is, no transition is possible.

We can express reachability of a deadlock state as the existence of a state with the dual property

$$G(N \perp \land \neg \text{terminal}$$
Termination and finiteness

A transition system or Kripke structure is called *terminating*, if every computation in it leads to a terminal state.

Termination for a Kripke structure is equivalent to the finiteness of all computation paths.

But a computation path is finite if and only if it contains a deadlock state.

Therefore, the following formula expresses that the computation tree is finite:

$$F \neg N \perp$$

(provided that this formula holds on every path).
Fairness

“The system must from time to time pass through a state which satisfies some property”

This is called a fairness constraint, and computations satisfying fairness constraints are called fair.

“The dispenser contains a drink infinitely often”

If the fairness property is expressed by a formula $\phi$, means that $A$ holds infinitely often on all paths.

$$G F \phi$$
Responsiveness

“Whether every request is eventually acknowledged”

\[ A \text{ (request } \rightarrow \text{ N F ack)} : \]
If we also want that request should remain true until it is acknowledged, responsiveness can be expressed by the formula

\[ A \text{ (request } \rightarrow \text{ (request U ack))} \]

We can also require that the request formula and the acknowledgement formula be mutually exclusive,

\[ A \text{ (request } \rightarrow \text{ ((request } \wedge \neg \text{ack) U (\neg request } \wedge \text{ack))}) \]
Alternation

Let \( \pi = s_0; s_1; s_2; \ldots \) be a path. We claim that the formula

\[ P \land A (P \leftrightarrow \neg X P) \]

"P is true at the even states \( s_0, s_2, s_4; \ldots \) and false at the odd states \( s_1, s_3, s_5 \) on this path".

\( \pi_0 \models A (P \leftrightarrow \neg X P), \Rightarrow \forall i, \pi_i \models P \text{ iff } \pi_{i+1} \models \neg P \). By \( \pi_i \models P \) means that \( \pi_0 \models P \), so P is true at the even states.
Introduction to NuSMV

NuSMV is a symbolic model checker developed by ITC-IRST and UniTN with the collaboration of CMU and UniGE.

The NuSMV project aims at the development of a state-of-the-art model checker that:

- Is robust, open and customizable;
- Can be applied in technology transfer projects;
- Can be used as research tool in different domains.
- NuSMV is OpenSource:

  - Is developed by a distributed community,
  - Has a “Free Software” license.
Introduction to NuSMV

NuSMV is a reimplementation and extension of SMV.

NuSMV started in 1998 as a joint project between ITC-IRST and CMU:

- the starting point: SMV version 2.4.4.
- SMV is the first BDD-based symbolic model checker (McMillan, 90).

NuSMV version 1 has been released in July 1999.

- limited to BDD-based model checking
- extends and upgrades SMV along three dimensions:
  - functionalities (LTL, simulation)
  - architecture
  - implementation

Results:

- used for teaching courses and as basis for several PhD theses
- interest by industrial companies and academics
The NuSMV 2 project started in September 2000 with the following goals:

- Introduction of SAT-based model checking
- OpenSource licensing
- Larger team (Univ. of Trento, Univ. of Genova, ...)

NuSMV 2 has been released in November 2001.

- First freely available model checker that combines BDD-based and SAT-based techniques
- Extended functionalities wrt NuSMV 1

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

MODULE main
VAR
    b0 : boolean;

ASSIGN
    init(b0) := 0;
    next(b0) := !b0;

An SMV program consists of:

- Declarations of the state variables (b0 in the example); the state variables determine the state space of the model.
- Assignments that define the valid initial states (init(b0) := 0).
- Assignments that define the transition relation (next(b0) := !b0).
Declaring state variables
The SMV language provides booleans, enumerative and bounded integers as data types:

**boolean:**

```plaintext
VAR
  x : boolean;
```

**enumerative:**

```plaintext
VAR
  st : {ready, busy, waiting, stopped};
```

**bounded integers (intervals):**

```plaintext
VAR
  n : 1..8;
```
MODULE main
VAR
   b0 : boolean;
   b1 : boolean;
ASSIGN
   init(b0) := 0;
   next(b0) := !b0;
   init(b1) := 0;

Remarks:
   + The new state space is the artesian product of the ranges of
     the variables.
   + Synchronous composition between the “subsystems” for b0 and b1.
Declaring the set of initial states

• For each variable, we constrain the values that it can assume in the initial states.

\[ \text{init(<variable>)} := \text{<simple_expression>} ; \]

• <simple expression> must evaluate to values in the domain of <variable>.

• If the initial value for a variable is not specified, then the variable can initially assume any value in its domain.
Introduction to NuSMV

Arithmetic operators: + - * / mod - (unary)

Comparison operators: = != > < <= >=

Logic operators: & | xor ! (not) -> <->

Conditional expression:
  case
    c1 : e1; if c1 then e1 else if c2 then e2 else if ... else en
    c2 : e2;
    ...
    1 : en;
  esac

Set operators:
  \{v1,v2,...,vn\} (enumeration) in (set inclusion) union (set union)
Expressions
Expressions in SMV do not necessarily evaluate to one value. In general, they can represent a set of possible values.

\[
\text{init(var)} := \{a,b,c\} \cup \{x,y,z\};
\]

The meaning of := in assignments is that the lhs can assume non-deterministically a value in the set of values represented by the rhs.

A constant c is considered as a syntactic abbreviation for \{c\} (the singleton containing c).
Declaring the transition relation

The transition relation is specified by constraining the values that variables can assume in the *next state*.

\[
\text{next(<variable>) := <next_expression> ;}
\]

<next expression> must evaluate to values in the domain of <variable>.

<next expression> depends on “current” and “next” variables:

\[
\text{next(a) := \{a, a+1\};}
\]
\[
\text{next(b) := b + (next(a) - a) ;}
\]

If no next() assignment is specified for a variable, then the variable can evolve non deterministically, i.e. it is unconstrained.

Unconstrained variables can be used to model non-deterministic *inputs* to the system.
Introduction to NuSMV

MODULE main
VAR
  b0 : boolean;
  b1 : boolean;
ASSIGN
  init(b0) := 0;
  next(b0) := !b0;
  init(b1) := 0;
  next(b1) := (!b0 & b1) | (b0 & !b1);
Specifying normal assignments

Normal assignments constrain the current value of a variable to the current values of other variables.

They can be used to model outputs of the system.

\[ \text{<variable>} := \text{<simple_expression>} ; \]

<simple_expression> must evaluate to values in the domain of the <variable>. 

Introduction to NuSMV
MODULE main
VAR
    b0 : boolean;
    b1 : boolean;
ASSIGN
    init(b0) := 0;
    next(b0) := !b0;
    init(b1) := 0;
    next(b1) := ((!b0 & b1) | (b0 & !b1));
    out := b0 + 2*b1;
The modulo 4 counter with reset
The counter can be reset by an external “uncontrollable” reset signal.

MODULE main
VAR
  b0 : boolean;
  b1 : boolean;
  reset : boolean;
  out : 0..3;
ASSIGN
  init(b0) := 0;
  next(b0) := case
    reset = 1 : 0;
    reset = 0 : !b0;
  esac;
  init(b1) := 0;
  next(b1) := case
    reset : 0;
    1 : ((!b0 & b1) | (b0 & !b1));
  esac;
  out := b0 + 2*b1;
An SMV program can consist of one or more *module declarations.*

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;

Modules are instantiated in other modules. The instantiation is performed inside the VAR declaration of the parent module.

In each SMV specification there must be a module main. It is the top-most module.

All the variables declared in a module instance are visible in the module in which it has been instantiated via the dot notation (e.g., m1.out, m2.out).
Module declarations may be *parametric*.

```plaintext
MODULE mod(in)
  VAR out: 0..9;
  ...
MODULE main
  VAR m1 : mod(m2.out);
    m2 : mod(m1.out);
  ...
```

*Formal parameters* (in) are substituted with the *actual parameters* (m2.out, m1.out) when the module is instantiated.

Actual parameters can be any legal expression.

Actual parameters are passed by reference.
The SMV language allows for the specification of different kinds of properties:

- Invariants,
- CTL formulas,
- LTL formulas.

Specifications can be added in any module of the program. Each specification is verified separately by NuSMV.
In the SMV language:
  • Different kinds of properties are allowed:

Properties on the reachable states
  • \textit{invariants} (INVARSPEC)

Properties on the computation paths (linear time logics):
  • \textit{LTL} (LTLSPEC)
LTL properties are specified via the keyword LTLSPEC:

\[
\text{LTLSPEC <ltl_expression>}
\]

A state in which \( \text{out} = 3 \) is eventually reached

\[
\text{LTLSPEC } \text{F out} = 3
\]

Condition \( \text{out} = 0 \) holds until \( \text{reset} \) becomes false

\[
\text{LTLSPEC } (\text{out} = 0) \text{ U } (!\text{reset})
\]

Even time a state with \( \text{out} = 2 \) is reached, a state with \( \text{out} = 3 \) is reached afterwards.

\[
\text{LTLSPEC } \text{G (out} = 2 \text{ } \rightarrow \text{ F out} = 3)
\]