

A glimpse at nuXmv

Luca Geatti

- NUXMV: is a symbolic model checker for the analysis of synchronous finite-state and infinite-state systems
- state-of-the-art algorithms:
 - For the finite-state case:
 - BDD-based model-checking, like its predecessor nuSMV.
 - strong verification engine based on modern SAT-based algorithms
 - For the infinite-state case: SMT-based verification techniques, implemented through a tight integration with MathSAT5.
- download it and try it!

<https://nuxmv.fbk.eu/>

Today:

- modeling and specification languages
- simulation
- model checking

Manual: <https://es-static.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf>

Modeling and Specification languages

Modeling language

- SMV language: Symbolic Model Verifier
 - introduced in 1993 in the seminal paper “Symbolic model checking: 10^{20} states and beyond”
- allows for the description of:
 - synchronous and asynchronous systems
 - networks/products of subsystems
 - non-deterministic behaviors
 - modular nature (very close to OO programming)
- SMV file = symbolic representation of a transition system (aka Kripke structure)

- State Variables
 - **Boolean**: `boolean`
 - **enum** : $\{item_1, item_2, \dots, item_n\}$
 - `integer : int`
 - ... a lot of others ...
- Input Variables
 - they are variables "controlled" by the environment
 - we can observe their value but...
 - we can **not** constrain their value in anyway

SMV - Transition Relation

- Initial states
 - any Boolean formula over the set of state variables
 - it is specified by the keyword **INIT**
- Transition Relation
 - any Boolean formula over the following set:

$$\mathcal{V} := \{v \mid v \text{ is a state or input variable}\} \\ \cup \\ \{\text{next}(v) \mid v \text{ is a state variable}\}$$

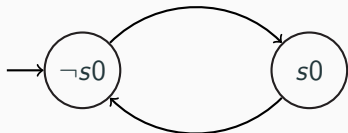
- it is specified by the keyword **TRANS**

- SMV allows also:
 - all arithmetic operations (addition, multiplication, etc)
 - trigonometric functions
 - bitwise operations

An alternative way:

- `ASSIGN init(v) := ...`
- `ASSIGN next(v) := ...`

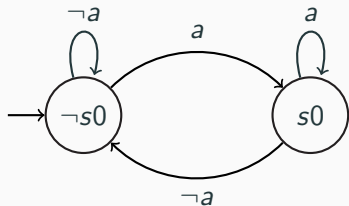
Example - Simple automaton



```
MODULE main
VAR
    s0 : boolean;
INIT
    !s0;
TRANS
    s0 <-> next(!s0);
```

Each of the 2^n assignments to the n state variables corresponds to a state of the explicit transition system.

Example with input variables



In this example, you can think of **variables** as **letters** of the alphabet of the automaton.

```
MODULE main
```

```
IVAR
```

```
  a : boolean;
```

```
VAR
```

```
  s0 : boolean;
```

```
ASSIGN
```

```
  init(s0) := FALSE;
```

```
  next(s0) := case
```

```
    !s0 & a      : TRUE;
```

```
    !s0 & !a    : FALSE;
```

```
    s0 & a      : TRUE;
```

```
    s0 & !a    : FALSE;
```

```
  esac;
```

- Mainly LTL and CTL
- ... but also:
 - past operators
 - PSL
 - real-time CTL
 - ...

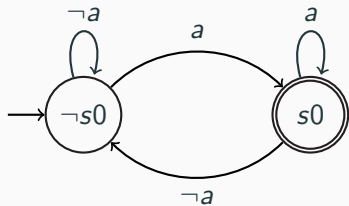
- LTL syntax:

$$\begin{aligned} \phi := & p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid X\phi \mid \phi_1 \mathcal{U} \phi_2 \\ & \mid F\phi \mid G\phi \mid \phi_1 \mathcal{R} \phi_2 \end{aligned}$$

- in SMV with the keyword **LTLSPEC**
 - LTLSPEC ltl_expr;
 - LTLSPEC NAME name_expr := ltl_expr;

Example - Simple DFA

\mathcal{A} :



$\tau \models F(s0)$ iff $\tau \in \mathcal{L}(\mathcal{A})$.

```
MODULE main
```

```
IVAR
```

```
  a : boolean;
```

```
VAR
```

```
  s0 : boolean;
```

```
ASSIGN
```

```
  init(s0) := FALSE;
```

```
  next(s0) := case
```

```
    !s0 & a      : TRUE;
```

```
    !s0 & !a    : FALSE;
```

```
    s0 & a      : TRUE;
```

```
    s0 & !a    : FALSE;
```

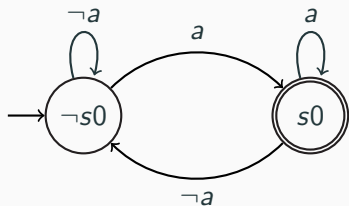
```
  esac;
```

```
LTLSPEC
```

```
  NAME final_dfa := F(s0)
```

Example - Simple Büchi automaton

\mathcal{A} :



$\tau \models GF(s0)$ iff $\tau \in \mathcal{L}(\mathcal{A})$.

```
MODULE main
```

```
IVAR
```

```
  a : boolean;
```

```
VAR
```

```
  s0 : boolean;
```

```
ASSIGN
```

```
  init(s0) := FALSE;
```

```
  next(s0) := case
```

```
    !s0 & a      : TRUE;
```

```
    !s0 & !a    : FALSE;
```

```
    s0 & a      : TRUE;
```

```
    s0 & !a    : FALSE;
```

```
  esac;
```

```
LTLSPEC
```

```
  NAME final_buchi :=
```

```
    GF(s0)
```

Simulation

- Simulation generates a trace (or a set of traces) of the SMV model.
- It can be used, for example,
 - for exploring different behaviors of the model
 - for checking if the model is an accurate representation of reality
- Simulation is different from model checking: it is not exhaustive.

These commands are prerequisites for all the other commands:

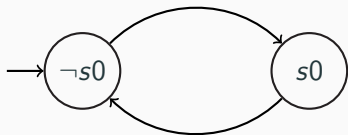
- `set_input_file file_name`: sets the file containing the model
- `go`: it parses the model file, it populates all the necessary data structures like BDD, etc.
- `go_bmc`: similar to the previous command
- `reset`: undo the effects of all the commands

Commands:

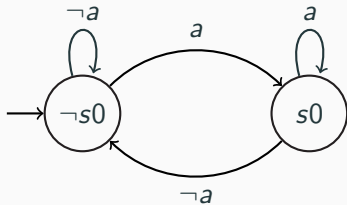
- `pick_state -v -i`: it picks an initial state for the trace
 - `-v`: verbose
 - `-i`: interactive mode, the user can choose the state from a set of possibilities
- `simulate -v -i -k 5`
 - `-k`: length of the trace

Examples

Without input variables:



With input variables:

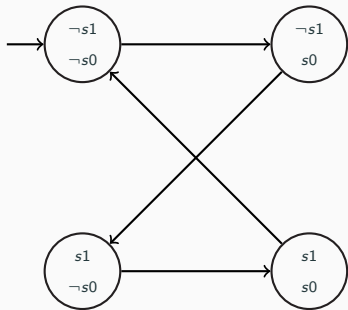


Model Checking

Plethora of commands for model checking:

- BDD-based model checking:
 - `check_ltlspec`
 - Burch, Jerry R., et al. "Symbolic model checking: 10^{20} states and beyond." (1992)
- SAT-based model checking:
 - BMC
 - `check_ltlspec_bmc`
 - Biere, Armin, et al. "Bounded model checking." (2003).
 - K-Liveness
 - `check_ltlspec_ic3`
 - Claessen, Koen, and Niklas Sörensson. "A liveness checking algorithm that counts." (2012)
 - IC3
 - `check_invar_ic3`
 - Bradley, Aaron R. "SAT-based model checking without unrolling." (2011)
 - it is tailored for *invariant* properties, that is, of type $G(\alpha)$

Example - Modulo 4 counter



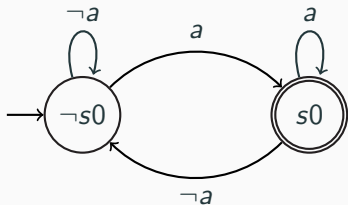
- $\phi_1 := GF(s0 \wedge s1)$ ✓
- $\phi_2 := FG(\neg s0 \wedge \neg s1)$ ✗
- $\phi_2 := G(s1 \rightarrow s0)$ ✗ : invariant spec, we can use IC3

Example - Simple Büchi automata

- we want to check the emptiness of the Büchi automaton \mathcal{A} :

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \emptyset$$

\mathcal{A} :

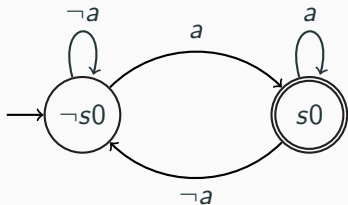


Example - Simple Büchi automata

- we want to check the emptiness of the Büchi automaton \mathcal{A} :

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \emptyset$$

\mathcal{A} :



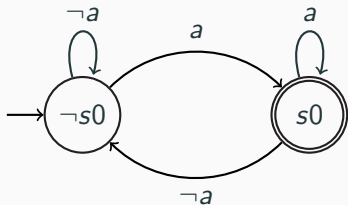
- how can we check it?

Example - Simple Büchi automata

- we want to check the emptiness of the Büchi automaton \mathcal{A} :

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \emptyset$$

\mathcal{A} :



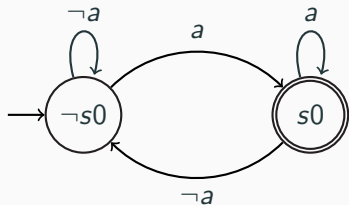
- how can we check it?
- ... with model checking?

Example - Simple Büchi automata

- we want to check the emptiness of the Büchi automaton \mathcal{A} :

$$\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \emptyset$$

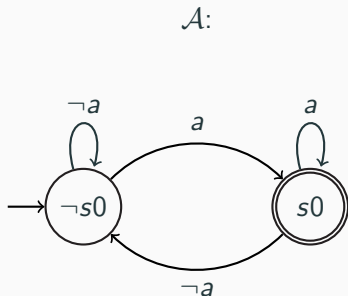
\mathcal{A} :



- how can we check it?
- ... with model checking?
- We reduce the emptiness problem to the following model-checking problem:

$$\mathcal{A} \stackrel{?}{\models} A(\text{FG}\neg s0)$$

Example - Simple Büchi automata



- it holds that:

$$\mathcal{A} \not\models A(FG\neg s0)$$

\Leftrightarrow

$$\mathcal{A} \models E(GFs0)$$

\Leftrightarrow

there exists an accepting run

\Leftrightarrow

$$\mathcal{L}(\mathcal{A}) \neq \emptyset$$

Appendix

A Three-Bit Counter

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

SPEC  AG AF bit2.carry_out

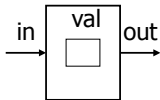
MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := FALSE;
  next(value) := value xor carry_in;
DEFINE
  carry_out := value & carry_in;
```



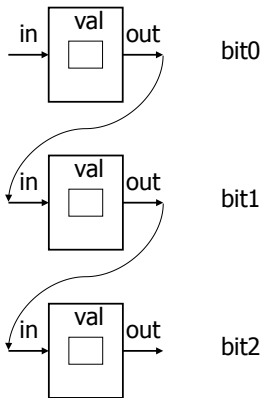
$value + carry_in \bmod 2$



module declaration



module instantiations



AG AF bit2.carry_out is true

