Learning outcomes

After this lecture you should:

- Be familiar with features of the SMV modelling language
- Be able to check simple LTL specification of models with NuSMV
- Be able to interpret an SMV counterexample trace

NB: all content in this lecture is non-examinable this year

An open-source model checker

NuSMV is a state-of-the-art model checker:

- Freely available as pre-built binaries for Windows, Linux, and Mac
- Also available in source form
- Good documentation, and tutorial material

A re-implementation of the SMV model checker:

- Was revolutionary in applying new techniques to model checking
- Could handle large models
- Was popular in semiconductor industry via Cadence SMV

See NuSMV homepage to download: http://nusmv.fbk.eu/
NuSMV’s components

NuSMV consists of two components:

• An implementation of SMV modelling language for describing finite state models
• Implementations of LTL, CTL, and PSL model checking algorithms

PSL = logic for verifying clocked hardware
We focus on LTL here

SMV modelling language

One-bit toggle

SMV models are:

• Split into modules
• Distinguished module called main, entry point similar to Java

NuSMV models declare “state variables” with associated types
Assignments constrain initial states, and describe transitions:

• \texttt{init(bit) := FALSE} is an initial assignment to bit
• \texttt{next(bit) := !bit} dictates how bit evolves

Some notes

MODULE main
VAR
  bit : boolean;
ASSIGN
  init(bit) := FALSE;
  next(bit) := !bit;
Built-in types

SMV has a number of built-in types:

- **boolean** has the values **TRUE** and **FALSE**
- **1..8** denotes a bounded interval of integer values
- **array 0..2 of boolean** denote a 2-element array

User-defined enumerations are also possible: e.g. \{R, Y, G, B\}

Assignments and non-determinism

Assignments can be made via *init* and *next*

If either one is omitted:

- Assignment is non-deterministic
- Value picked from possible values based on type

Useful for modelling environment, introducing abstraction, etc.

Assignments induce equations used to build underlying model

To ensure model exists, equations are syntactically restricted:

- Variables may only be assigned once,
- No loops within assignments

Case statements

In case expression:

```
case
  request : busy;
  TRUE      : {ready,busy};
esac;
```

Cases evaluated sequentially, first matching case is taken

Cases need not be deterministic:

- \{**ready, busy**\} means status evolves to **ready** or **busy** non-deterministically
- Singleton **busy** is syntactic sugar for \{**busy**\}
SMV’s interactive mode

Saving example in `short.smv`

Load model in NuSMV’s interactive mode:

```bash
$ ./bin/NuSMV -int short.smv
```

Ask NuSMV to set itself up ready for use:

```bash
NuSMV > go
```

This compiles model, sets up variables, and so on

```bash
NuSMV > pick_state
```

Asks NuSMV to pick initial state consistent with assignments

---

Generating traces

Asking NuSMV to randomly generate a trace of length 3:

```bash
NuSMV > simulate -v -r -k 3
```

Produces:

```
-> State: 1.1 <-
  request = FALSE
  status = ready
-> State: 1.2 <-
  request = FALSE
  status = busy
-> State: 1.3 <-
  request = FALSE
  status = busy
-> State: 1.4 <-
  request = FALSE
  status = ready
```

---

LTL model checking

Property:

*It is always the case that if a request is made, then eventually the system will be busy.*

Rendered in LTL:

```latex
\Box (request \rightarrow \Diamond (status = busy))
```

Rendered in SMV’s LTL assertion language:

```latex
G(request \rightarrow F \text{status=busy})
```

---

Checking the property

Using the `check_ltlspec` command:

```bash
NuSMV > check_ltlspec -p ``G(request -> F status=busy)''
```

NuSMV checks the property against the model, and produces:

```
-- specification G(request -> F status = busy) is true
```
Checking a non-property

Non-property:

It is always the case that if there is no request, then there will be one eventually.

Rendered in LTL:

\( \neg \text{request} \rightarrow F \text{request} \)

Rendered in SMV's LTL assertion language:

\( G(!\text{request} \rightarrow F \text{request}) \)

Semaphore: user module

\[
\text{MODULE } \text{user}(\text{semaphore}) \\
\text{VAR} \\
\quad \text{state} : \{\text{idle, entering, critical, exiting}\}; \\
\text{ASSIGN} \\
\quad \text{init}(\text{state}) := \text{idle}; \\
\quad \text{next}(\text{state}) := \\
\quad \quad \text{case} \\
\quad \quad \quad \text{state} = \text{idle} : \{\text{idle, entering}\}; \\
\quad \quad \quad \text{state} = \text{entering} \& \neg \text{semaphore} : \text{critical}; \\
\quad \quad \quad \text{state} = \text{critical} : \{\text{critical, exiting}\}; \\
\quad \quad \quad \text{state} = \text{exiting} : \text{idle}; \\
\quad \quad \quad \text{TRUE} : \text{state}; \\
\quad \quad \text{esac}; \\
\quad \text{next}(\text{semaphore}) := \\
\quad \quad \text{case} \\
\quad \quad \quad \text{state} = \text{entering} : \text{TRUE}; \\
\quad \quad \quad \text{state} = \text{exiting} : \text{FALSE}; \\
\quad \quad \quad \text{TRUE} : \text{semaphore}; \\
\quad \quad \text{esac}; \\
\]

Semaphore: main module

\[
\text{MODULE } \text{main} \\
\text{VAR} \\
\quad \text{semaphore} : \text{boolean}; \\
\quad \text{process1} : \text{process user(}\text{semaphore})\text{);} \\
\quad \text{process2} : \text{process user(}\text{semaphore})\text{);} \\
\text{ASSIGN} \\
\quad \text{init}(\text{semaphore}) := \text{FALSE}; \\
\]

Checking the non-property

Once again, using the \texttt{check_ltlspec} command:

\[
\text{NuSMV} > \text{check_ltlspec} -p `G(!\text{request} \rightarrow F \text{request})' \\
\]

NuSMV produces a counter-example, indicating property is false:

\[
\begin{aligned}
\text{-- specification } & \ G \ (!\text{request}\rightarrow\text{F request}) \text{ is false} \\
\text{...}
\text{Trace Type: Counterexample} \\
\text{-- Loop starts here} \\
\rightarrow \text{State: 2.1 <-} \\
\quad \text{request} = \text{FALSE} \\
\quad \text{status} = \text{ready} \\
\rightarrow \text{State: 2.2 <-}
\end{aligned}
\]

i.e. a run of the system where a request is never made is permissible
Parameterised modules

SMV allows models to be split into submodules

These modules may be parameterised

Formal parameters are passed when module is instantiated

Actual parameters may be any legal SMV expression

Processes

In `main` module we instantiate `user` module twice

We have marked each instantiation with the `process` keyword

This has the effect of introducing "interleaving" concurrency:

- One process is chosen non-deterministically
- All of its assignments are executed in parallel
- Another process is chosen non-deterministically
- And so on...

A built-in scheduler picks a process to run at each step

Two concurrent processes trying to enter critical section

---

Example trace

-> State: 1.1 <-
    semaphore = FALSE
    process1.state = idle
    process2.state = idle

-> Input: 1.2 <-
    _process_selector_ = process1
    running = FALSE
    process2.running = FALSE
    process1.running = TRUE

-> State: 1.2 <-
    semaphore = FALSE
    process1.state = entering
    process2.state = idle

...  

Transitions are interleaved by scheduler picking a process to execute

---

Counter: counter cell module

```plaintext
MODULE counter(increment)
VAR
    digits : 0..9;
ASSIGN
    init(digits) := 0;
    next(digits) := increment ? (digits + 1) mod 10 : digits;
DEFINE
    overflow := digits = 9;
```
Counter: main module

```plaintext
MODULE main
VAR
  counter1 : counter(TRUE);
  counter2 : counter(counter1.overflow);
result : 0..99;
ASSIGN
  result := counter1.digits + counter2.digits * 10;
LTLSPEC
  G(result = 1)
```

Definitions and immediate assignments

In counter cell we made use of **DEFINE**:

- Introduces new definition
- Can be thought of as a macro: `digits = 9` will replace `overflow` throughout

Further, made use of an immediate assignment:

```plaintext
  result := counter1.digits + counter2.digits * 10
```

Constrains value of `result` using `counter1.digits` and `counter2.digits`

Inline LTL specifications

LTL specification can be embedded within a model

Need not be provided interactively within NuSMV shell

Use LTLSPEC block to provide an LTL formula as specification

Run NuSMV in batch mode to check property:

```bash
$ ./bin/NuSMV counter.smv
```

Gives same output as interactive mode

Case study
Recall: clocked circuit

R is a register, with initial value 0

Recall: pictorial model

Modelling a register

\[
\text{MODULE register(input)}
\]
\[
\text{VAR}
\]
\[
\text{last : boolean;}
\]
\[
\text{current : boolean;}
\]
\[
\text{ASSIGN}
\]
\[
\text{init(current) := FALSE;}
\]
\[
\text{init(last) := input;}
\]
\[
\text{next(current) := last;}
\]
\[
\text{next(last) := input;}
\]
\[
\text{DEFINE}
\]
\[
\text{output := current;}
\]

Registers have a “memory” of last input value, and initially read 0

Modelling gates

\[
\text{MODULE or_gate(input1, input2)}
\]
\[
\text{DEFINE}
\]
\[
\text{output := input1 | input2;}
\]

\[
\text{MODULE xor_gate(input1, input2)}
\]
\[
\text{DEFINE}
\]
\[
\text{output := input1 xor input2;}
\]

\[
\text{MODULE inverter(input1)}
\]
\[
\text{DEFINE}
\]
\[
\text{output := !input1;}
\]

Note: use of modules overkill here
Modelling circuit

```plaintext
MODULE main
VAR
    x_input : boolean;
    y_output : boolean;
    shared_wire : boolean;

    OR : or_gate(x_input, shared_wire);
    R : register(OR.output);
    XOR : xor_gate(x_input, shared_wire);
    NOT : inverter(XOR.output);
ASSIGN
    shared_wire := R.output;
    y_output := NOT.output;

Note: shared_wire to break cycle in circuit diagram
```

Circuit properties

Was our pictorial diagram of circuit behaviour correct?

\[
\text{LTLSPEC} \quad G(x_{\text{input}} \land R.\text{output} \rightarrow y_{\text{output}})
\]

\[
\text{LTLSPEC} \quad G(!x_{\text{input}} \land !R.\text{output} \rightarrow y_{\text{output}})
\]

\[
\text{LTLSPEC} \quad G(x_{\text{input}} \land R.\text{output} \rightarrow y_{\text{output}})
\]

\[
\text{LTLSPEC} \quad G(x_{\text{input}} \land !R.\text{output} \rightarrow !y_{\text{output}})
\]

Computer says yes

NuSMV claims all properties are true

Circuit properties

Does setting the input bit to high always imply the register output bit will eventually read low?

\[
\text{LTLSPEC} \quad G(x_{\text{input}} \rightarrow F !R.\text{output})
\]

Is output bit Y set infinitely often?

\[
\text{LTLSPEC} \quad G F y_{\text{output}}
\]
Computer says no, for both
(And also produces counterexample traces)

Summary

In this lecture you have:

• Become familiar with NuSMV, a state-of-the-art open source model checker
• Become familiar with NuSMV’s interactive and batch modes
• Been introduced to major elements of the SMV specification language
• Seen some simple models written in SMV
• Seen some simple verifications/counter examples of LTL specifications