

Introduction to SMV

New Symbolic Model Verifier

- Originally, SMV by Ken McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, 1993.
- **NuSMV**: Re-implementation at FBK-IRST (Trento).
- You can get source & binary at:
<http://nusmv.fbk.eu/>
- Finite-state Systems specified in a specialized language
- Specifications given as LTL or CTL formulas + Fairness
- Internal representation using BDDs.
- Automatically checks specification or gives a counterexample

Language Characteristics

- Allows description of synchronous and asynchronous systems
- Modular and hierarchical descriptions
- Finite data types: Boolean, bounded integers, scalars (enumerations), arrays.
- Nondeterminism

A Sample NuSMV Program

MODULE main

VAR

request: boolean;

state: {ready, busy};

ASSIGN

init(state) := ready;

next(state) :=

case

state=ready & request: busy;

TRUE: {ready, busy};

esac;

LTLSPEC G(request -> F (state = busy))

Variable Assignments

- Assignment to **initial** state:

`init(value) := 0;`

- Assignment to next state (**transition** relation)

`next(value) := value + carry_in mod 2;`

- Assignment to current state (**invariant**)

`carry_out := value & carry_in;`

- Use either init-next or invariant - never both
- NuSMV is a parallel assignment language

The Case Expression

- **case** is an *expression*, not a statement
- Guards are evaluated *sequentially*.
- The *first* one that is *true* determines the resulting value
- If *none* of the guards are *true*, an *arbitrary* valid value is returned
 - Always use an *else guard*!

Nondeterminism

- Completely *unassigned variable* can model *unconstrained input*.
- $\{val_1, \dots, val_n\}$ is an expression taking on any of the given values *nondeterministically*.
 - Use **union** when you have expressions rather than values
- **Nondeterministic** choice can be used to:
 - Model an implementation that has *not* been *refined* yet
 - *Abstract* behavior

Types

- Boolean
 - 1 is true and 0 is false
- Integer
 - Integers typically range from $-2^{32}+1$ to $2^{32}-1$
- Enumeration

- **VAR**

```
a : {red, blue, green};  
b : {1, 2, 3};  
c : {1, 5, 7};
```

ASSIGN

```
next(b) := case  
    b < 3 : b + 1;  
    TRUE : 1;  
esac;
```

- *Numerical operations* must be properly guarded

Types

- Bit-vector
 - `word[]` e.g., `bv : word[4]` declared a vector of 4 bits.
 - Bit selection operators on `word[]` type: e.g., `0b6_011001[4:1]` gives constant `0b4_1100`
 - Shifting operators on `word[]` type: e.g. `0b6_011001<<2` results in the constant `0b6_100100`
 - Logical, relational and arithmetic operators can be applied to the `word[]` type
- Array
 - `a`: array 0..3 of boolean;
 - `b`: array 10..20 of {OK, y, z};
 - `c` : array 1..8 of array -1..2 of `word[4]`;
 - Limited use in expressions e.g.: `c[3][-1] & 0b4_1100`

ASSIGN and DEFINE

- VAR a : boolean;
ASSIGN $a := b \mid c$;
 - declares a new state variable a
 - becomes part of invariant relation
- DEFINE $d := b \mid c$;
 - is effectively a macro definition, each occurrence of d is replaced by $b \mid c$
 - no extra BDD variable is generated for d
 - the BDD for $b \mid c$ becomes part of each expression using d

Next (transition relation)

- Expressions can refer to the value of a variable in the *next* state
- Examples:
 - **VAR** a,b : boolean;
ASSIGN
 next(b) := !b;
 a := next(b);
 - **ASSIGN** next(a) := !next(b)
(a is the negation of b, except for the initial state)
 - Disclaimer: different (Nu)SMV versions differ on this

Circular definitions

- ... are not allowed!
- This is illegal:
 - `a := next(b);`
`next(b) := c;`
`c := a;`
- This is o.k.
 - `init(a) := 0;`
`next(a) := !b;`

`init(b) := 1;`
`next(b) := !a;`

`init(c) := 0;`
`next(c) := a & next(b);`

Modules and Hierarchy

- Modules can be instantiated many times, each instantiation creates a copy of the local variables
- Each program has a module `main`
- Scoping
 - Variables declared outside a module can be passed as parameters
 - Internal variables of a module can be used in enclosing modules (referred to with the *complex identifier* `submodel.varname`).
 - The *full identifier* is the *complex identifier* of a module's variable as seen from the MAIN module
- Parameters are passed by reference.

Scoping

MODULE main

VAR

a : boolean;

b : foo;

c : moo;

MODULE foo

VAR q : boolean;

e : moo;

MODULE moo

DEFINE f := 0 < 1;

MODULE not_used

VAR

n : boolean;

t : used;

MODULE used

VAR

k : boolean;

Variable	Full Identifier
a	a
q	b.q
e	b.e
f	b.e.f, c.f
n,t,k	NONE

```
MODULE main
```

```
VAR
```

```
  bit0 : counter_cell(1);
```

```
  bit1 : counter_cell(bit0.carry_out);
```

```
  bit2 : counter_cell(bit1.carry_out);
```

```
LTLSPEC G (F bit2.carry_out)
```

```
LTLSPEC G (F !bit2.carry_out)
```

```
MODULE counter_cell(carry_in)
```

```
VAR  value : boolean;
```

```
ASSIGN
```

```
  init(value) := 0;
```

```
  next(value) := (value + carry_in) mod 2;
```

```
DEFINE carry_out := value & carry_in;
```

Module Composition

- **Synchronous** composition
 - All assignments are executed in parallel and synchronously.
 - A single step of the resulting model corresponds to a step in each of the components.
- **Asynchronous** composition (inteleaving)
 - A step of the composition is a step by exactly one process.
 - Variables, not assigned in that process, are left unchanged.

Asynchronous Composition

```
MODULE main
```

```
VAR
```

```
  gate1: process inverter(gate3.output);
```

```
  gate2: process inverter(gate1.output);
```

```
  gate3: process inverter(gate2.output);
```

```
LTLSPEC G (F gate1.output)
```

```
LTLSPEC G (F !gate1.output)
```

```
MODULE inverter(input)
```

```
VAR  output: boolean;
```

```
ASSIGN
```

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

```
  next(output) := !input;
```

Counterexamples

-- specification $G (F \text{ !gate1.output})$ is false
-- as demonstrated by the following execution

state 2.1:

gate1.output = FALSE

gate2.output = FALSE

gate3.output = FALSE

state 2.2:

[executing process gate1]

-- loop starts here --

state 2.3:

gate1.output = TRUE

[stuttering]

Fairness

- **FAIRNESS** formulae [**JUSTICE** formulae]
 - Assumed to be true infinitely often
 - Model checker only explores paths satisfying fairness constraint
 - Each fairness constraint must be true infinitely often
- If there are no fair paths
 - All existential formulas are false
 - All universal formulas are true
- **FAIRNESS** running
running is an implicit boolean variable of every process instance of a module. It is **true** when the module instance is **scheduled for execution**.

Counter Revisited

```
MODULE main
```

```
VAR
```

```
    count_enable: boolean;
```

```
    bit0 : counter_cell(count_enable);
```

```
    bit1 : counter_cell(bit0.carry_out);
```

```
    bit2 : counter_cell(bit1.carry_out);
```

```
SPEC G (F bit2.carry_out)
```

```
FAIRNESS count_enable
```

```
[...]
```

Example: Client & Server

MODULE client (**ack**)

VAR

state : {idle, requesting};

req : **boolean**;

ASSIGN

init(**state**) := idle;

next(**state**) :=

case

state =idle : {idle, requesting};

state =requesting & **ack** : idle;

TRUE : **state**;

esac;

req := (**state**=requesting);

Example: Client & Server

MODULE server (**req**)

VAR

state : {idle, pending, acking};

ack : boolean;

ASSIGN

next(**state**) :=

case

state=idle & **req** : pending;

state=pending : {pending, acking};

state=acking & **req** : pending;

state=acking & !**req** : idle;

TRUE : **state**;

esac;

ack := (**state** = acking);

Is the specification true?

MODULE main

VAR

c : client(**s.ack**);

s : server(**c.req**);

LTLSPEC G (c.req → F s.ack)

- Need fairness constraint:

- Suggestion:

FAIRNESS s.ack

- Why is this bad?

- Solution:

FAIRNESS !(state=pending)

in **server** spec.

Running NuSMV (interactively)

- **NuSMV -int**
 - Runs NuSMV in interactive mode
- **read_model -i <filename>**
 - Reads a system spec. from file
- **go**
 - Builds the internal representation of the model
- **check_fsm**
 - Checks whether the transition relation is total

NuSMV Options

- `compute_reachable`
 - computes set of **reachable states** first
 - the *model checking algorithm* traverses only the set of reachable states instead of complete state space.
 - useful if reachable state space is a small fraction of total state space
- `print_reachable_states`
 - prints out the number of reachable states
- `check_ltlspec [check_ctlspec]`
 - checks all the LTL properties [CTL properties] included in the file spec. of the system

Variable Reordering

- Variable reordering is crucial for small BDD sizes and speed.
- Generally, variables which are related need to be close in the ordering.
- **encode_variables -i <filename>**
 - Input BDD variable ordering from a given file.
- **write_order -o <filename>**
 - Output BDD variable ordering to a given file.
- **dynamic_var_ordering [-e sift] [-d]**
 - Enable/disable automatic variable reordering