

Symbolic Model Checkers

(Based on [Clarke *et al.* 1999])

Wei-Shao, Tang

SVVRL
Dept. of Information Management
National Taiwan University

May 14, 2015

- 🌐 Introduction to SMV and NuSMV
- 🌐 Input Language
- 🌐 Examples: Mutual Exclusion and FutureBus+
- 🌐 LTL, CTL, and BMC in NuSMV
- 🌐 References

Symbolic Model Verifier (SMV)

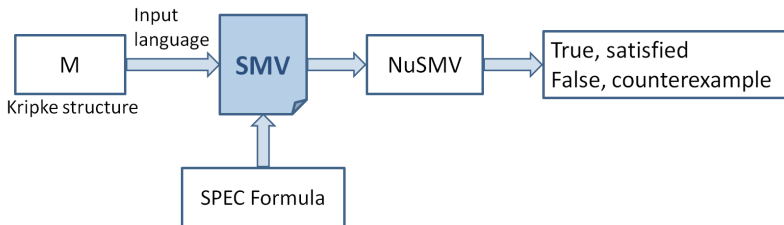
- 🌐 SMV is a tool to check finite state system that **satisfies specifications** in CTL.
- 🌐 SMV uses the **BDD-based** symbolic model checking algorithm.
- 🌐 The first model checker based on BDDs.
- 🌐 The language component of SMV is used to describe complex finite-state system.
- 🌐 The primary purpose of the SMV input language is to describe the transition relation of a finite **Kripke structure**.

NuSMV(1/1)

- 🌐 NuSMV is a new symbolic model checker, reimplementing and extending CMU SMV.
- 🌐 NuSMV 2 is [Open Source](#) and the latest version is NuSMV 2.5.4 (Oct 28, 2011)
- 🌐 NuSMV allows for the representation of [synchronous](#) and [asynchronous](#) finite state systems.
- 🌐 The analysis of specifications expressed in Computation Tree Logic ([CTL](#)) and Linear Temporal Logic ([LTL](#)), using [BDD-based](#) and SAT-based (Mini-Sat) model checking techniques.





NuSMV(2/2)

- A SMV file includes the **input language** for description of finite state machine and SPEC formulas that be used to verify our desired properties.
- NuSMV Work flow diagram:




- 🌐 Introduction to SMV and NuSMV
- 🌐 **Input Language**
- 🌐 Examples: Mutual Exclusion and FutureBus+
- 🌐 LTL, CTL, and BMC in NuSMV
- 🌐 References

Modules



-  User can decompose the descriptions of finite-state systems into **modules**.
-  Individual modules can be instantiated multiple times, and modules can reference **variables** declared in other modules.
-  Modules can have parameters, which in turn may be state components, expressions, or other modules.
-  Modules can also contain **fairness constraints**.

- 🌐 Synchronous and interleaved composition
 - ☀️ SMV modules can be composed either synchronously or using interleaving.
 - ☀️ In a **synchronous** component, a single step in the composition corresponds to a single step in each of the components.
 - ☀️ With **interleaving**, a single step in the composition represents a step by exactly one component. (use keyword **process**)

Nondeterministic transitions

-  **Nondeterminism** can reflect **actual choice** in the actions of the system being modeled, or it can be used to describe a more abstract model.

Transition relations









-  It can be specified explicitly in terms of boolean relations on the **current and next state** values of state variables,
-  or implicitly as a set of parallel **assignment** statements.

A Simple Example

- The following is a simple example that illustrates the **basic concepts**.

```
MODULE main
VAR
  request : boolean;
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
    state = ready & request : busy;
  TRUE : {ready, busy};
  esac;
SPEC
  AG(request -> AF state = busy)
```

Types Overview(1/2)

-  boolean
-  integer
-  enumeration
 -  symbolic enum
ex: {stopped, running, waiting}
 -  integers-and-symbolic enum
ex: {-1, 1, waiting}
-  word: are used to model vector of bits (booleans) which allow bitwise logical and arithmetic operations
 -  unsigned word [•]
 -  signed word [•]

Types Overview(2/2)

- array: are declared with lower and upper bound for the index, and the type of the elements in the array.
ex: array 0..3 of boolean
array 1..8 of array -1..2 of unsigned word[5]
- set: are used to identify expressions representing a set of values.
 - boolean set
 - integer set
 - symbolic set
 - integers-and-symbolic set

Expressions(1/4)

🌐 Constant Expressions

```
constant ::boolean_constant
          | integer_constant
          | symbolic_constant
          | word_constant
          | range_constant
```

🌐 `word_constant :: 0 [word_sign_specifier]`
`word_base [word_width] _ word_value`

ex: `0sb5_10111` has type `signed word[5]`

🌐 `range constant :: integer_number .. integer_number`

Expressions(2/4)

Basic Expressions

```
basic_expr :: constant
            |variable_identifier
            |define_identifier
            |! basic_expr
            |basic_expr & basic_expr
            |basic_expr | basic_expr
            |basic_expr -> basic_expr
            |basic_expr = basic_expr
            |basic_expr ? basic_expr : basic_expr
            |basic_next_expr
            |case_expr
            |{ set_body_expr }
            :
```

Expressions(3/4)

Case Expressions





```
case_expr ::  
  case  
    expr_a1 : expr_b1 ;  
    expr_a2 : expr_b2 ;  
    :  
    expr_an : expr_bn ;  
  esac
```

If-Then-Else Expressions

```
cond_expr ? basic_expr1 : basic_expr2
```



Expressions(4/4)

Set Expressions

-  Defining a set of boolean, integer and symbolic enum values
-  There cannot be a set of sets in NuSMV
-  Be created with the union operator
-  `set_body_expr :: basic_expr`
`| set_body_expr , basic_expr`

ex: expression {exp1, exp2, exp3} is equivalent to exp1 **union** exp2
union exp3

Next Expressions

-  Refer to the values of variables in the next state
-  `basic_next_expr :: next (basic_expr)`

Statement declaration - Variable(1/3)

- A variable can be an input, a frozen, or a state variable.
- Type Specifiers

```
type_specifier :: simple_type_specifier
                | module_type_specifier
```

```
simple_type_specifier :: boolean
                    | word [ basic_expr ]
                    | unsigned word [ basic_expr ]
                    | signed word [ basic_expr ]
                    | { enumeration_type_body }
                    | basic_expr .. basic_expr
                    | array basic_expr .. basic_expr
                    of simple_type_specifier
```

Statement declaration - Variable(2/3)

State Variables

- ☀ A state of the model is an assignment of values to a set of state and frozen variables.
- ☀ `var_declaration :: VAR var_list`
`var_list :: identifier : type_specifier ;`
`| var_list identifier : type_specifier ;`
- ☀ Example: `VAR a : {stopped, running, finished};`

Input Variables

- ☀ IVARs (input variables) are used to label transitions of the Finite State Machine
- ☀ `ivar_declaration :: IVAR simple_var_list`
`simple_var_list :: identifier : simple_type_specifier ;`
`| simple_var_list identifier :`
`simple_type_specifier ;`
- ☀ Example: `IVAR b : {TRUE, FALSE};`

Statement declaration - Variable(3/3)

🌐 Frozen Variables

- ☀️ FROZENVAR s (frozen variables) are variables that retain their initial value throughout the evolution of the state machine

- ☀️ `frozenvar_declaration :: FROZENVAR simple_var_list`

- ☀️ Semantic meaning:

```
ASSIGN next(a) := a;
```

- ☀️ Example:

```
FROZENVAR a : boolean;
```


```
VAR b : boolean;
```

```
ASSIGN
```

```
next(a) := b; -- illegal
```

```
a := b; -- illegal
```

ASSIGN Constraint

```
 assign_constraint :: ASSIGN assign_list
assign_list :: assign ;
               | assign_list assign ;
assign :: complex_identifier := simple_expr
        | init ( complex_identifier ) := simple_expr
        | next ( complex_identifier ) := next_expr
```

Example of **ASSIGN**

```
ASSIGN
init(turn) := 0;
next(turn) :=
case
  turn = turn0 & state0 = critical:!turn;
  TRUE: turn;
esac;
```

Statement declaration - Constraint(3/5)

TRANS Constraint

- The transition relation of the model is a set of current state/next state pairs
- The transition relation is the conjunction of all of TRANS
- `trans_constraint :: TRANS next_expr [;]`

INIT Constraint

- The set of initial states of the model is determined by a boolean expression under the **INIT**
- The expression doesn't contain the `next()` operator.
- The initial set is the conjunction of all of **INIT**
- `init_constrain :: INIT simple_expr [;]`

Example:

```
INIT output = 0
```

```
TRANS
```





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

```
  | next(output)=output
```

Statement declaration - Constraint(4/5)

INVAR Constraint

```
invar_constraint :: INVAR simple_expr [;]
```

-  The set of invariant states can be specified using a boolean expression under the INVAR keyword.
-  The expression doesn't contain the next () operator.
-  The invariant is the conjunction of all of INVAR.
-  Example:

```
INVAR x = y + 1
```

Statement declaration - Constraint(5/5)

- 🌐 Semantically assignments can be expressed using other kinds of constraints
 - ☀️ `ASSIGN a := exp;`
is equivalent to `INVAR a = exp;`
 - ☀️ `ASSIGN init(a) := exp;`
is equivalent to `INIT a = exp;`
 - ☀️ `ASSIGN next(a) := exp;`
is equivalent to `TRANS next(a) = exp;`

Statement declaration - Spec & Fairness(1/2)

- 🌐 SPEC declaration
decl :: "SPEC" ctlform
 - ☀️ A CTL formula doesn't contain next() operator.
 - ☀️ A CTL formula return a value 0 or 1.
 - ☀️ The specification is the conjunction of all of SPEC.

- 🌐 FAIRNESS constraint declaration

```
fairness_constraint ::  
FAIRNESS simple_expr [;]  
| JUSTICE simple_expr [;]  
| COMPASSION ( simple_expr , simple_expr ) [;]
```

Example of SPEC and FAIRNESS

SPEC

$AG((s0 = \text{trying}) \rightarrow AF (s0 = \text{critical}))$

FAIRNESS $!(s0 = \text{critical})$

Statement declaration - DEFINE & MODULE(1/2)

DEFINE Declarations

```
define_declaration ::= DEFINE define_body  
define_body ::= identifier := simple_expr ;  
| define_body identifier := simple_expr ;
```

MODULE Declaratios

```
module ::= MODULE identifier [( module_parameters )]  
[module_body]
```

Statement declaration - DEFINE & MODULE(2/2)

Example of MODULE and DEFINE

```
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;
```

Statement declaration - main & identifier(1/2)

References to Module Components

- Both of variable identifiers and define identifiers are complex identifiers

```
complex_identifier :: identifier
    | complex_identifier . identifier
    | complex_identifier [ simple_expression ]
    | self
```

A Program and the main Module

- There must be one module with the name `main` and no formal parameters.

```
program :: module
    | module_list module
```

Statement declaration - main & identifier(2/2)

- 🌐 Example of main and identifiers.

```
MODULE main
... VAR
  a : bar;
  m : foo(a);
...
MODULE bar
  VAR
    q : boolean;
    p : boolean;
MODULE foo(c)
  DEFINE
    flag := c.q | c.p;
```

Statement declaration - CTL(1/2)

CTL Specifications

```
ctl_specification ::= CTLSPEC ctl_expr [;]
                   | SPEC ctl_expr [;]
                   | CTLSPEC NAME name := ctl_expr [;]
                   | SPEC NAME name := ctl_expr [;]
```

Statement declaration - CTL(2/2)

🌐 A CTL formula has the syntax

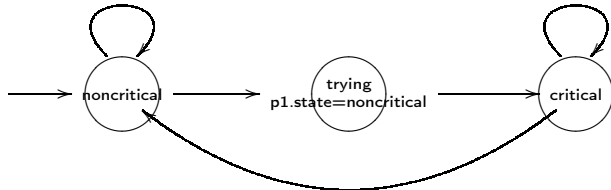
```
ctl_expr ::= simple_expr
          | ! ctl_expr
          | ctl_expr & ctl_expr
          | ctl_expr | ctl_expr
          | ctl_expr -> ctl_expr
          | ctl_expr <-> ctl_expr
          | EG ctl_expr
          | EX ctl_expr
          | EF ctl_expr
          | AG ctl_expr
          | AX ctl_expr
          | AF ctl_expr
          | E [ ctl_expr U ctl_expr ]
          | A [ ctl_expr U ctl_expr ]
```


Agenda

- 🌐 Introduction to SMV and NuSMV
- 🌐 Input Language
- 🌐 Examples: Mutual Exclusion and FutureBus+
- 🌐 LTL, CTL, and BMC in NuSMV
- 🌐 References

Mutual Exclusion Problem(1/7)

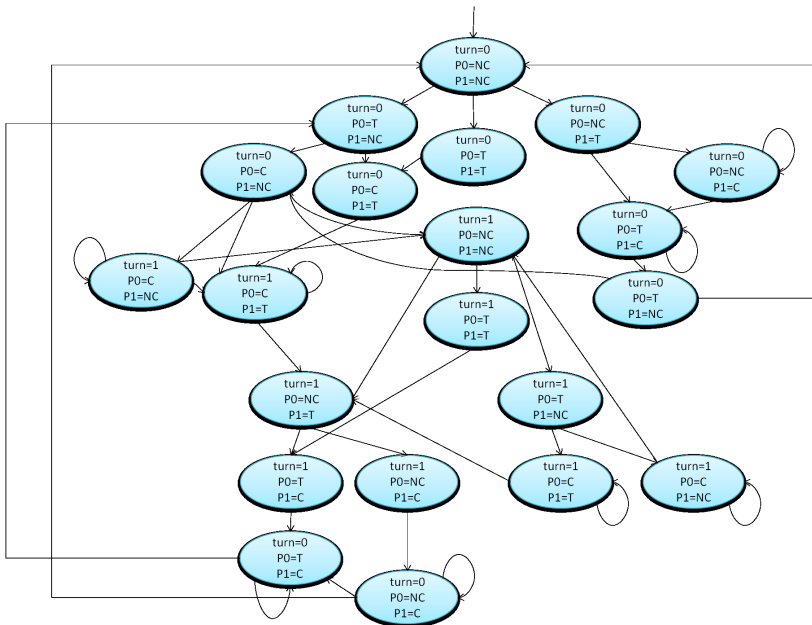
- The goal of this program is to exclude the possibility that both processes are in their **critical regions at the same time**.
- A process which wants to enter its critical region will **eventually be able to enter**.
- Each process in one of three region: **noncritical**, **trying**, **critical**.



Mutual Exclusion Problem(2/7)

- Initially, both processes are in their noncritical regions.
- A process is in trying region and the other is in noncritical region, the first process can immediately enter its critical region.
- If both processes are in their trying regions, the boolean variable **turn is used to determine which process enters its critical region**.
 - if $turn = 0$ then process 0 can enter and $turn := !turn$.
 - if $turn = 1$ then process 1 can enter and $turn := !turn$.
- We assume that a process must eventually leave its critical region.
- It may remain in its noncritical region forever.

Mutual Exclusion Problem(3/7)



Code of Mutual Exclusion

```
1  MODULE main --two process mutual exclusion
2  VAR
3  s0: {noncritical, trying, critical};
4  s1: {noncritical, trying, critical};
5  turn: boolean;
6  pr0: process prc(s0, s1, turn, 0);
7  pr1: process prc(s1, s0, turn, 1);
8  ASSIGN
9  init(turn) := 0;
```

Mutual Exclusion Problem(4/7)

- 🌐 **Module definitions** begin with the keyword `MODULE`.
 - ☀ The module `main` is top-level module. (line 1)
 - ☀ The module `prc` has formal parameter `state0`, `state1`, `turn`, `turn0`. (line 19)
- 🌐 **Variables** are declared using `VAR`.
 - ☀ i.e., `turn` is a boolean variable, while `s0` and `s1` are variables which can have one of three region. (line 3-5)
 - ☀ It's also used to instantiate other modules. (line 6-7)
 - ☀ The keyword `process` is used in both cases, the global model is constructed by **interleaving** steps from `pr0` and `pr1`.

Code of Mutual Exclusion(cont'd)

```
19 MODULE prc(state0, state1, turn, turn0)
20 ASSIGN
21   init(state0) := noncritical;
22   next(state0) :=
23     case
24       (state0= noncritical):{trying,noncritical};
25       (state0= trying)&(state1= noncritical): critical;
26       (state0= trying)&(state1= trying)&(turn = turn0):
27         critical;
28       (state0= critical) : {critical,noncritical};
29   1:state0;
30   esac;
```

Code of Mutual Exclusion(cont'd)

```
30 next(turn) :=
31 case
32   turn = turn0 & state0 = critical: !turn;
33   1: turn;
34 esac;
```


Mutual Exclusion Problem(5/7)

- 🌐 The ASSIGN statement is used to define the initial states and transitions of the model.
 - ☀ i.e.,the initial value of variable `turn` is 0. (line 9)
 - ☀ The value of the variable `state0` and `turn` in the next state is given by the case statement. (line 23-29) (line 31-34)
 - ☀ The value of a case statement is determined by evaluating the clauses within the statement in sequence.
 - ☀ When a set expression is assigned to a variable, the value of variable is **chosen nondeterministically** from the set.

Code of Mutual Exclusion(cont'd)

```
10 FAIRNESS    !(s0 = critical)
11 FAIRNESS    !(s1 = critical)
12 SPEC       EF((s0 = critical) & (s1 = critical))
13 SPEC       AG((s0 = trying) -> AF (s0 = critical))
14 SPEC       AG((s1 = trying) -> AF (s1 = critical))
15 SPEC       AG((s0 = critical) -> A[(s0 = critical) U
16            (!s0 = critical) & !E[!(s1 = critical) U
17            (s0 = critical)]])
18 SPEC       AG((s1 = critical) -> A[(s1 = critical) U
19            !(s1 = critical) & !E[!(s0 = critical) U
20            (s1 = critical)]])
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35 FAIRNESS    running
```

Mutual Exclusion Problem(6/7)

- 🌐 The FAIRNESS statements are **fairness constrains**.
 - ☀️ Fairness constrains (line10-11) are used to **prevent a process remain in its critical region forever**.
- 🌐 The CTL properties to be verified are given as SPEC statements.
 - ☀️ The first specification checks for a violation of the **mutual exclusion requirement**.(line 12)
 - ☀️ The second and third check that a process which wants to enter its critical region will **eventually be able to enter**.(line 13-14)
 - ☀️ The last two specifications check whether processes must **strictly alternate entry into their critical regions**.(line 15-17)

Mutual Exclusion Problem(7/7)

Result:

- ☀ $EF((s0 = \text{critical}) \ \& \ (s1 = \text{critical}))$ is **false**
- ☀ $AG((s0 = \text{trying}) \rightarrow AF(s0 = \text{critical}))$ is **true**
- ☀ $AG((s1 = \text{trying}) \rightarrow AF(s1 = \text{critical}))$ is **true**
- ☀ $AG((s0 = \text{critical}) \rightarrow A[(s0 = \text{critical})..])$ is **false**
- ☀ $AG((s1 = \text{critical}) \rightarrow A[(s1 = \text{critical})..])$ is **false**

The **output** note following:

- ☀ mutual exclusion is not violated,
- ☀ absence of starvation is true,
- ☀ strict alternation of critical region is false.

SMV produced **counterexample computation paths** in the false cases.

Counterexample

🌐 Counterexample for **strict alternation** of critical regions.

```
-- specification AG (s0 = critical -> A(...) is false
-- as demonstrated by the following execution sequence
state 2.1: s0 = noncritical
           s1 = noncritical
           turn=0
state 2.2: [executing process pr0]
state 2.3: [executing process pr0]
           s0 = trying
state 2.4: s0 = critical
state 2.5: [executing process pr0]
state 2.6: s0 = noncritical
           turn = 1
state 2.7: [executing process pr0]
state 2.8: [executing process pr0]
           s0 = trying
state 2.9: s0 = critical
```

A Realistic Example: *Futurebus+*

- 🌐 The formalization and verification of the cache coherence protocol
 - ☀️ draft IEEE *Futurebus+* standard (IEEE Standard 896.1-1991).
- 🌐 A precise model of the protocol was constructed in SMV language and model checking was used to show that it satisfied a formal **specification of cache coherence**.
- 🌐 A number of errors and ambiguities were discovered.
- 🌐 This experience demonstrates that hardware description and model checking techniques can be used to help design real industrial standards.
- 🌐 We will briefly discuss about the design and protocol of Futurebus+, then introduce two simple examples.

- 🌐 *Futurebus+* is a bus architecture for high-performance computers.
- 🌐 The cache coherence protocol used in *Futurebus+* is required to **insure consistency of data** in hierarchical systems composed of many processors and caches interconnected by multiple bus segments.
- 🌐 The model is highly **nondeterministic**, both to reduce the complexity of verification and to cover allowed design choices.
- 🌐 The model for the cache coherence protocol consists of 2300 lines of SMV code.

Design of *Futurebus+*

- 🌐 *Futurebus+* maintains coherence by having the individual caches **snoop**, or observe, all bus transaction and update their status.
- 🌐 Coherence across buses is maintained using bus **bridges**.
- 🌐 Special **agents** at the end of the bridges represent remote caches and memories.
- 🌐 The protocol uses **split transaction** to increase performance.
- 🌐 **Split transaction**: The completion of the transaction is delayed and the bus is freed; at some later time, an explicit response is issued to complete the transaction.
- 🌐 This facility makes it possible to service local requests while remote requests are being processed.

Design of *Futurebus+* (cont'd)

- 🌐 **Cache line**: a series of consecutive memory locations that is treated as a unit for coherence purposes.
- 🌐 We are interested in *cache modules* that represents a cache/processor pair and *shared memory modules*.
- 🌐 Each cache module in the system is required to keep an **attribute** for the **cache line**; the attribute represents the read and write access the cache has to the line.
- 🌐 The attributes specified by the *Futurebus+* protocol are:
 - ☀ **invalid**
 - ☀ **shared unmodified**
 - ☀ **exclusive unmodified**
 - ☀ **exclusive modified**

Design of *Futurebus+* (cont'd)

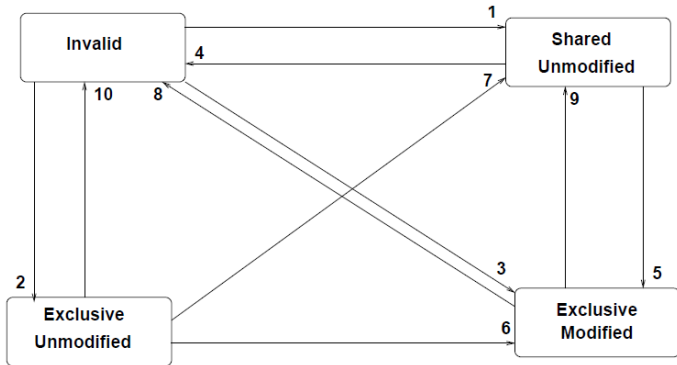
- The standard defines a number of **transactions** that relate to the movement of the data lines.
- **Read Shared**: This transaction is initiated by a cache which wishes to obtain read access to the data line
- **Read Modified**: is initiated by a cache who wishes to obtain read/write access to the data line
- **Invalidate**: is initiated by a cache who has read access to the data line and wishes to obtain write access to the line

Design of *Futurebus+*(cont'd)

- 🌐 **Copyback**: is initiated by a cache has modified the data line and wishes to evict the line from its cache memory.
- 🌐 **Shared Response**: is initiated by a cache who has forced another module to go into a requester state. This response is sharable, others may snarf it.
- 🌐 **Modified Response**: is initiated by a cache has forced another module to go into a requester state. This response is not sharable.

Design of *Futurebus+* (cont'd)

- Transition diagram between line attribute in response to transactions.



Source: Esser. "Verification of the Futurebus+ Cache Coherence protocol: A case study in model checking", 2003

Design of *Futurebus+* (cont'd)

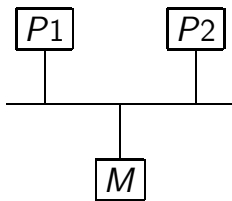
- 1 The module completed a read shared transaction that was snarfed by another module, or it has snarfed the completed read shared transaction of another module.
- 2 Completed a read shared transaction that was not snarfed by another module
- 3 Completed a read modified transaction
- 4 The module may voluntarily clear the cache of a line, or the module did not snarf read shared transaction belonging to another module, or another module initiated read modified or invalidate transaction.
- 5 Completed an invalidate transaction

Design of *Futurebus+*(cont'd)

- 6 The module may change an exclusive unmodified line to exclusive modified at any time without a bus transaction.
- 7 The module may change the line state to shared-unmodified without a bus transaction, or the module snarfed the read shared transaction of another module.
- 8 Removed the line from the cache (after performing a copyback transaction)
- 9 The module performed a copyback transaction and kept a copy of the line.
- 10 Removed the line from the cache, or the module did not snarf the read share transaction of another module, or another module initiated a read modified transaction.

Example of *Futurebus+*: Single bus

- We consider some example transactions for a single *cache line* in the two-processor system.
- Initially, neither processor has a copy of the line in its cache.
- All processor are in the **invalid** state.

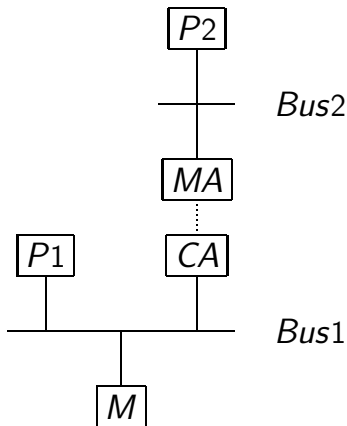


Example of *Futurebus+*: Single bus(cont'd)

- 🌐 P1 issues a **read-shared** transaction to obtain a readable copy of the data from **M(memory)**.
- 🌐 P2 snoops this transaction, and it also can obtain a readable copy, this is called **snarfing**.
- 🌐 If P2 snarfs, both caches contain a **shared-unmodified** copy.
- 🌐 Next, P1 decides to write, and issues an **invalidate** transaction on the bus.
- 🌐 P2 snoops this transaction, and delete the copy.
- 🌐 Final, P1 has an **exclusive-modified** copy of the data.

Two-bus Example

- Initially, both processor caches are in the **invalid state**.
- Each processor doesn't have a copy in its cache.



Two-Bus Example(cont'd)

- 🌐 P2 issues a **read-modified** to obtain a writable copy, then MA(memory agent) splits the transaction, for it must get the data from M.
- 🌐 The command is passed to CA(cache agent), and CA issues the read-modified on bus 1.
- 🌐 M supplies the data to CA, which in turn passes it to MA.
- 🌐 MA issues a **modified-response** on bus 2 to complete the split transaction.

Two-Bus Example(cont'd)

- Suppose now that P1 issues a **read-shared** command.
- CA, knowing that a remote cache has an exclusive-modified copy, **intervenes** in the transaction to indicate that it will supply the data, and splits the transaction.
- CA passes the read-shared to MA, which issues it.
- P2 intervenes and supplies the data to MA, which passes it to CA.
- CA performs a **shared-response** transaction which complete the read-shared issued by P1.

- 🌐 First, a number of the low-level details dealing with how modules communicate were eliminated.
 - ☀️ The most significant simplification was to use a model in which one step corresponds to one transaction.
- 🌐 Second, it was used to reduce the size of some parts of the system.
 - ☀️ E.g., only transactions involving a **single cache line** were considered.
 - ☀️ The data were reduced to single bit.

Simplifications(cont'd)

- Third, it involved eliminating the **read-invalid** and **write-invalid** commands.
 - These commands are used in DMA transfers to and from memory.
- Last, it involved using nondeterminism to simplify the models of some of the components.
 - Processor are assumed to issue read and write requests for a given cache line nondeterministically.
 - Responses to split transactions are assumed to be issued after arbitrary delays.
 - Finally, the model of a bus bridge is highly nondeterministic.

Cache Model

```
1 next(state) :=
2   case
3   CMD=none:
4     case
5     state=share-unmodified:
6       case
7       requester=exclusive: share-unmodified;
8       1: invalid, shared-unmodified;
9       esac;
10    state=exclusive-unmodified: invalid, shared-unmodified,
11    exclusive-unmodified, exclusive-modified;
12    1: state;
13    esac;
14  :
```

Cache Model(cont'd)

- State components with (CMD, SR, TF) denote bus signals visible to the cache, and components with (state, tf) are under the control of the cache.
- This part specifies what happen when an idle cycle occurs.
- If the cache has a **shared-unmodified** copy, then the line may be nondeterministically kicked out of the cache unless there is an outstanding request to change the line to exclusive-modified.
- If a cache has an **exclusive-unmodified** copy of the line, it may kick the line out of the cache or change it to exclusive-modified.

Cache Model(cont'd)

```
15 master:
16     case
17     CMD=read-shared:
18         case
19         state=invalid:
20             case
21                 !SR & !TF: exclusive-unmodified;
22                 !SR: shared-unmodified;
23                 1: invalid;
24             esac;
25         :
26     esac;
27 :
28 :
29 :
```


Cache Model(cont'd)

- 🌐 This part indicate how the cache line state is updated when the **cache issues a read-shared transition**.
- 🌐 This should only happen when the cache doesn't have a copy.
- 🌐 If the transaction is not split (!SR), then the data will be supplied to the cache.
- 🌐 Either no other caches will snarf the data (!TF), in which case the cache obtain an exclusive-unmodified copies.
- 🌐 If the transition is split, the cache line remains in the invalid state.

Cache Model(cont'd)

```
30  CMD=read-shared:
31      case
32      state = invalid, shared-unmodified:
33          case
34              !tf: invalid;
35              !SR: shared-unmodified;
36              1: state;
37          esac;

38      :
41  esac;
```

Cache Model(cont'd)

- 🌐 This part tells how caches respond when they observe another one issuing a read-shared transaction.
- 🌐 If the observing cache is either invalid or shared-unmodified, then it may indicate that it doesn't want a copy and the line becomes invalid.
- 🌐 Alternatively, it may assert τf and try to snarf the data. The transaction is not split (!SR) , the cache obtains a shared-unmodified copy.
- 🌐 Otherwise, the case stays in it current state.

Specifications

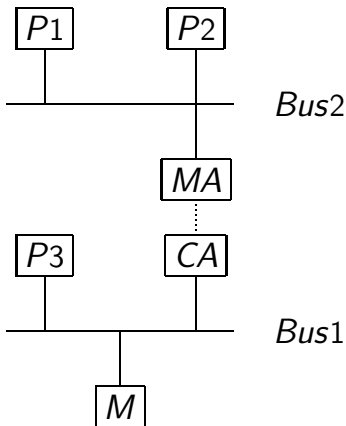
- 🌐 $AG(p1.writable \rightarrow \neg p2.readable)$
 - ☀️ If p1 is in the exclusive-modified state, p2 is in invalid.
- 🌐 $AG(p1.readable \wedge p2.readable \rightarrow p1.data = p2.data)$
 - ☀️ If two caches have copies, then they have the same data.
- 🌐 $AG(p.readable \wedge \neg m.memory-line-modified \rightarrow p.data = m.data)$
 - ☀️ If memory has an up-to-date data, then any cache that has a copy must agree with memory on the data.
- 🌐 $AG EF p.readable \wedge AG EF p.writable$
 - ☀️ This is used to check that it is always possible for a cache to get read or write access to the line.

Two of the errors

- 🌐 The **first error** occurs in the single bus protocol.
- 🌐 Initially, both caches are invalid.
- 🌐 P1 obtain an exclusive-unmodified copy.
- 🌐 Next, P2 issues a read-modified, which P1 splits for invalidation.
- 🌐 M supplies a copy to P2, which transitions to shared-unmodified.
- 🌐 At this point, P1, still having an exclusive-unmodified copy, transitions to exclusive-modified and writes the cache line.
- 🌐 P1 and P2 are **inconsistent**.
- 🌐 The bug can be fixed by requiring that P1 transition to the shared-unmodified state when it splits the read-modified for invalidation.

Two of the errors(cont'd)

- The **second error** occurs in the hierarchical configuration.
- P1, P2, and P3 all obtain share-unmodified copies.



Two of the errors(cont'd)

- 🌐 P1 issues an invalidate transaction that P2 and MA split.
- 🌐 P3 issues an invalidate that CA splits.
- 🌐 The bridge detects that an **invalidate-invalidate collision** has occurred.
- 🌐 The collision should be resolved by having MA invalidate P1.
- 🌐 When MA tries to do this, P2 asserts a busy signal on the bus.
- 🌐 MA observes this and acquires the **requester-waiting** attribute.

Two of the errors(cont'd)

- 🌐 P2 now finishes invalidating and issues a modified-response. This is split by MA because P3 still not invalid.
- 🌐 However, MA still maintains the requester-waiting attribute.
- 🌐 MA will not issue commands since it is waiting for a completed response, but no such response can occur.
- 🌐 There is a **deadlock**.
- 🌐 The deadlock can be avoided by having MA clear the requester-waiting attribute when it observe that P2 has finished invalidating.

- 🌐 Introduction to SMV and NuSMV
- 🌐 Input Language
- 🌐 Examples: Mutual Exclusion and FutureBus+
- 🌐 LTL, CTL, and BMC in NuSMV
- 🌐 References

LTL, CTL, and BMC in NuSMV


- 🌐 The main purpose of a model checker is to verify that a model satisfies a set of **desired properties** specified by the user.
- 🌐 In NuSMV, the specifications to be checked can be expressed in two different temporal logics: the **Computation Tree Logic (CTL)**, and the **Linear Temporal Logic (LTL)**.
- 🌐 CTL and LTL specifications are evaluated by NuSMV in order to determine their truth or falsity in the FSM
- 🌐 When a specification is discovered to be **false**, NuSMV constructs and **prints a counterexample**.

LTL Statement declaration

🌐 A LTL formula has the syntax

```
LTLexpr ::= LTLexpr
          | "!" LTLexpr
          | LTLexpr1 "&" LTLexpr2
          | LTLexpr1 "|" LTLexpr2
          | LTLexpr1 "->" LTLexpr2
          | LTLexpr1 "<->" LTLexpr2
Future operators
          | "X" LTLexpr
          | "G" LTLexpr
          | "F" LTLexpr
          | LTLexpr"U" LTLexpr
          | LTLexpr"V" LTLexpr
```

LTL Statement declaration(cont'd)

 A LTL formula has the syntax

LTLexpr :: Past operators

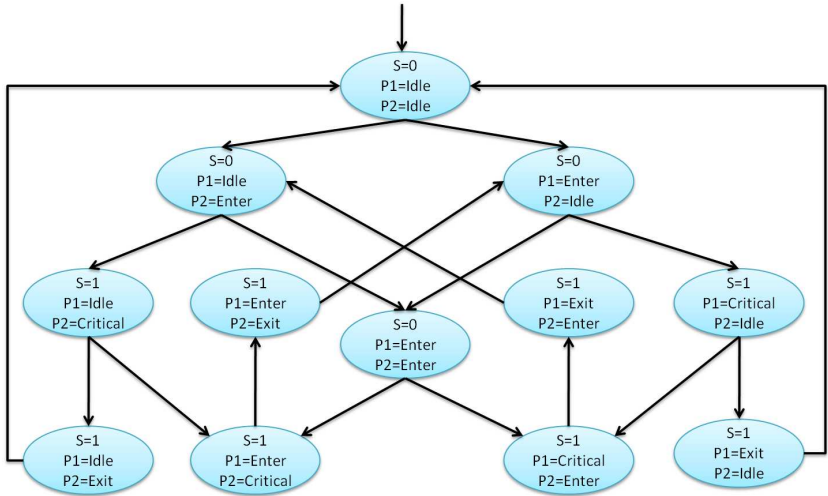
- | "Y" LTLexpr previous state
- | "Z" LTLexpr before
- | "H" LTLexpr historically
- | "O" LTLexpr once
- | LTLexpr"S" LTLexpr since
- | LTLexpr"T" LTLexpr triggered

LTL Statement declaration(cont'd)

- Y**: Previous state and not allowed using in the first state (will be FALSE).
- Z**: Previous state and allowed using in the first state (will be TRUE).
- H**: All states from begin to now.
- O**: If the state exists at least once in the states until now.
- S**: The latter expression has held once. And before it holds, the former expression have to always hold.
- T**: After the former expression holds, the latter expression have to always hold until now.

- Each process has **four state**: idle, entering, critical and exiting.
- The entering state indicate that the process wants to enter its critical region.
- If semaphore is 0, it goes to the critical, and sets semaphore to 1.
- In exiting state, the process sets semaphore to 0.

Semaphore(cont'd)



Code of Semaphore

```
1  MODULE main
2  VAR
3  semaphore : boolean;
4  proc1 : process user(semaphore);
5  proc2 : process user(semaphore);

6  ASSIGN
7  init(semaphore) := 0;
```


Code of Semaphore(cont'd)

```
8  MODULE user(semaphore)
9  VAR
10 state : {idle, entering, critical, exiting};
11 ASSIGN
12   init(state) := idle;
13   next(state) :=
14   case
15     state = idle: {idle, entering};
16     state = entering & !semaphore: critical;
17     state = critical: {critical, exiting};
18     state = exiting: idle;
19     1: state;
20   esac;
```

Code of Semaphore(cont'd)

```
21 next(semaphore) :=
22 case
23   state = entering: 1;
24   state = exiting: 0;
25   1: semaphore;
26 esac;

27 FAIRNESS
28 running
```

CTL Specification of Semaphore

- 🌐 proc1 and prco2 are not at the same time in the critical state.

SPEC

$AG!(\text{proc1.state=critical} \ \& \ \text{proc2.state=critical})$

- 🌐 If porc1 wants to enter its critical state, it eventually does.

SPEC

$AG(\text{proc1.state=entering} \ \rightarrow \ AF \ \text{proc1.state=critical})$

LTL Specification of Semaphore

- The two process cannot be in the critical region at the same time.

LTLSPEC

$G!(\text{proc1.state=critical} \ \& \ \text{proc2.state=critical})$

- A process wants to enter its critical session, it eventually does.

LTLSPEC

$G(\text{proc1.state=entering} \ \rightarrow \ F \ \text{proc1.state=critical})$

- A process enters its critical session, it once want to do it.

LTLSPEC

$G(\text{proc1.state=critical} \ \rightarrow \ O \ \text{proc1.state=entering})$

Bounded Model Checking in NuSMV

- 🌐 Instruct NuSMV to run in BMC by using command-line option `-bmc`
- 🌐 In BMC mode NuSMV tries to find a **counterexample** of increasing length, and immediately stops when it succeeds, declaring that the formula is **false**.
- 🌐 If the maximum number of iterations is reached and no counterexample is found, then NuSMV exits, and the **truth of the formula is not decided**.
- 🌐 The maximum number of iterations can be controlled by using `bmc_length`.
- 🌐 The default value is 10.

Example of Bounded Model Checking

```
1  MODULE main
2  VAR
3      y : 0..15;
4  ASSIGN
5      init(y) := 0;
6  TRANS
7  case
8      y = 7 : next(y) = 0;
9      1     : next(y) = ((y + 1) mod 16);
10 esac
```

Checking LTL Specifications with BMC

- 🌐 Check the following LTL specification with BMC

LTLSPEC G (y=4 -> X y=6)

False

LTLSPEC !G F (y = 2)

False

LTLSPEC F (X y=8 | 0 y<3)

This formula can't be decided within 10 iterations

- 🌐 Introduction to SMV and NuSMV
- 🌐 Input Language
- 🌐 Examples: Mutual Exclusion and FutureBus+
- 🌐 LTL, CTL, and BMC in NuSMV
- 🌐 References

Reference

- 🌐 Clarke *et al.*, "*Model Checking Ch. 8*", 1999.
- 🌐 K.L. McMillan, "*The SMV system*", 2000.
- 🌐 Roberto *et al.*, "*NuSMV 2.5 Tutorial*", 2010
- 🌐 Roberto *et al.*, "*NuSMV 2.5 User Manual*", 2010
- 🌐 Clarke *et al.*, "*Verification of the Futurebus+ Cache Coherence Protocol*", 1995.
- 🌐 Robert Esser , "*Verification of the Futurebus+ Cache Coherence protocol: A case study in model checking*", 2003.