The SMV system*

for SMV version 2.5.4

K. L. McMillan
`mcmillan@cs.cmu.edu`

Original: February 2, 1992
Last updated: November 6, 2000

The SMV system is a tool for checking finite state systems against specifications in the temporal logic CTL. The input language of SMV is designed to allow the description of finite state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can readily specify a system as a synchronous Mealy machine, or as an asynchronous network of abstract, nondeterministic processes. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only data types in the language are finite ones – Booleans, scalars and fixed arrays. Static, structured data types can also be constructed. The logic CTL allows a rich class of temporal properties, including safety, liveness, fairness and deadlock freedom, to be specified in a concise syntax. SMV uses the OBDD-based symbolic model checking algorithm to efficiently determine whether specifications expressed in CTL are satisfied.

The primary purpose of the SMV input language is to describe the transition relation of a finite Kripke structure. Any expression in the propositional calculus can be used to describe this relation. This provides a great deal of flexibility, and at the same time a certain danger of inconsistency. For example, the presence of a logical contradiction can result in a deadlock – a state or states with no successor. This can make some specifications vacuously true, and makes the description unimplementable. While the model checking process can be used to check for deadlocks, it is best to avoid the problem when possible by using a restricted description style. The SMV system supports this by providing a parallel-assignment syntax. The semantics of assignment in SMV is similar to that of single assignment data flow languages. By checking programs for multiple parallel assignments to the same variable, circular assignments, and type errors, the interpreter insures that a program using only the assignment mechanism is implementable. Consequently, this fragment of the language can be viewed as a hardware description language, or a programming language. The SMV system is by no means the last word on symbolic model checking techniques, nor is it intended to be a complete hardware description language. It is simply an experimental tool for exploring

*LaTeX sources restored by Tatsuhiro Tsuchiya, Dept. of Informatics and Mathematical Science, Osaka University, Japan, `t-tutiya@ics.es.osaka-u.ac.jp`. Edited and updated by Sergey Berezin, Carnegie Mellon University, `berez@cs.cmu.edu` the possible applications of symbolic model checking to hardware verification.

This document describes the syntax and semantics of the SMV input language, and the function of the SMV model checker. It also describes some optional features of the model checker which can be used to fine tune the performance, and gives some examples of its application. All of the examples in this document are made available with the software. For a description of all the model checker options, see the UNIX programmer's manual entry for SMV, which is also included with the software.

1 The input language

Before delving into the syntax and semantics of the language, let us first consider a few simple examples that illustrate the basic concepts. Consider the following short program in the language.

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

The input file describes both the model and the specification. The model is a Kripke structure, whose state is defined by a collection of state variables, which may be of Boolean or scalar type. The variable `request` is declared to be a Boolean in the above program, while the variable state is a scalar, which can take on the symbolic values `ready` or `busy`. The value of a scalar variable is encoded by the interpreter using a collection of Boolean variables, so that the transition relation maybe represented by an OBDD. This encoding is invisible to the user, however.

The transition relation of the Kripke structure, and its initial state (or states), are determined by a collection of parallel assignments, which are introduced by the keyword ASSIGN. In the above program, the initial value of the variable `state` is set to `ready`. The next value of `state` is determined by the current state of the system by assigning it the value of the expression

```
  case
    state = ready & request : busy;
    1 : {ready,busy};
  esac;
```

The value of a case expression is determined by the first expression on the right hand side of a : such that the condition on the the left hand side is true. Thus, if `state =`

`ready & request` is true, then the result of the expression is `busy`, otherwise, it is the set {`ready,busy`}. When a set is assigned to a variable, the result is a non-deterministic choice among the values in the set. Thus, if the value of `status` is not `ready`, or `request` is false (in the current state), the value of `state` in the next state can be either `ready` or `busy`. Non-deterministic choices are useful for describing systems which are not yet fully implemented (*ie.*, where some design choices are left to the implementor), or abstract models of complex protocols, where the value of some state variables cannot be completely determined.

Notice that the variable `request` is not assigned in this program. This leaves the SMV system free to choose any value for this variable, giving it the characteristics of an unconstrained input to the system.

The specification of the system appears as a formula in CTL under the keyword `SPEC`. The SMV model checker verifies that all possible initial states satisfy the specification. In this case, the specification is that invariantly if `request` is true, then inevitably the value of `state` is `busy`.

The following program illustrates the definition of reusable modules and expressions. It is a model of a 3 bit binary counter circuit. Notice that the module name "`main`" has special meaning in SMV, in the same way that it does in the C programming language. The order of module definitions in the input file is inconsequential.

```
MODULE main
VAR
  bit0 : counter_cell(1);
  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) := 0;
  next(value) := value + carry_in mod 2;
DEFINE
  carry_out := value & carry_in;
```

In this example, we see that a variable can also be an instance of a user defined module. The module in this case is `counter_cell`, which is instantiated three times, with the names `bit0`, `bit1` and `bit2`. The counter cell module has one formal parameter `carry_in`. In the instance `bit0`, this formal parameter is given the actual value 1. In the instance `bit1`, `carry_in` is given the value of the expression `bit0.carry_out`. This expression is evaluated in the context of the main module. However, an expression of the form $a.b$ denotes component b of module a, just as if the module a were a data structure in a standard programming language. Hence, the `carry_in` of module `bit1` is the `carry_out` of module `bit0`. The keyword `DEFINE` is used to assign the expression `value & carry_in` to the symbol `carry_out`.

Definitions of this type are useful for describing Mealy machines. They are analogous to macro definitions, but notice that a symbol can be referenced before it is defined.

The effect of the DEFINE statement could have been obtained by declaring a variable and assigning its value, as follows:

```
VAR
  carry_out : boolean;
ASSIGN
  carry_out := value & carry_in;
```

Notice that in this case, the *current* value of the variable is assigned, rather than the next value. Defined symbols are sometimes preferable to variables, how ever, since they don't require introducing a new variable into the BDD representation of the system. The weakness of defined symbols is that they cannot be given values non-deterministically. Another difference between defined symbols and variables is that while variables are statically typed, definitions are not. This may be an advantage or a disadvantage, depending on our point of view.

In a parallel-assignment language, the question arises: "What happens if a given variable is assigned twice in parallel?" More seriously: "What happens in the case of an absurdity, like `a := a + 1;` (as opposed to the sensible `next(a) := a + 1;`)." In the case of SMV, the interpreter detects both multiple assignments and circular references in expressions, and treats these as semantic errors, even in the case where the corresponding system of equations has a unique solution. Another way of putting this is that there must be a total order in which the assignments can be executed which respects all of their data dependencies. The same logic applies to defined symbols. As a result, all legal SMV programs *can* be implemented.

By default, all of the assignment statements in an SMV program are executed in parallel and simultaneously. It is possible, however, to define a collection of parallel processes, whose actions are interleaved in the execution sequence of the program. This is useful for describing communication protocols, or asynchronous circuits, or other systems whose actions are not synchronized (including synchronous circuits with more than one clock region). This technique is illustrated by the following program, which represents a ring of three inverting gates.

```
MODULE main
VAR
  gate1 : process inverter(gate3.output) ;
  gate2 : process inverter(gate1.output) ;
  gate3 : process inverter(gate2.output) ;
SPEC
  (AG AF gate1.out) & (AG AF !gate1.out)

MODULE inverter(input)
VAR
  output : boolean;
ASSIGN
  init(output) := 0;
  next(output) := !input;
```

A *process* is an instance of a module which is introduced by the keyword `process`. The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Because the choice of the next process to execute is non-deterministic, this program models the ring of inverters independently of the speed of the gates. For each gate, the specification of this program states that the output of the gate oscillates (*ie.*, that its value is infinitely often zero, and infinitely often 1). In fact, this specification is false, since the system is not forced to eventually choose a given process to execute, hence the output of a given gate may remain constant, regardless of its input.

In order to force a given process to execute infinitely often, we can use a *fairness constraint*. A fairness constraint restricts the attention of the model checker to only those execution paths along which a given CTL formula is true infinitely of ten. Each process has a special variable called `running` which is true if and only if that process is currently executing. By adding the declaration

```
FAIRNESS
  running
```

to the module `inverter`, we can effectively force every instance of `inverter` to execute infinitely often, thus making the specification true.

One advantage of using processes to describe a system is that it allows a particularly efficient OBDD representation of the transition relation. We observe that the set of states reachable by one step of the program is the union of the sets of states reachable by each individual process. Hence, rather than constructing the transition relation of the entire system, we can use the transition relations of the individual processes separately and combine the results. This can yield a substantial savings in space in representing the transition relation. Occasionally , however, the fact that two processes cannot make simultaneous transitions leads to increased complexity in representing the set of states reachable by n steps.

The alternative to using processes to model an asynchronous circuit would be to have all gates execute simultaneously, but allow each gate the non-deterministic choice of evaluating its output, or keeping the same output value. Such a model of the inverter ring would look like the following:

```
MODULE main
VAR
  gate1 : inverter(gate3.output);
  gate2 : inverter(gate2.output);
  gate3 : inverter(gate1.output);
SPEC
  (AG AF gate1.out) & (AG AF !gate1.out)

MODULE inverter(input)
VAR
  output : boolean;
```

```
ASSIGN
  init(output) := 0;
  next(output) := !input union output;
```

The set union operator coerces its arguments to singleton sets as necessary. Thus, the next output of each gate can be either its current output, or the negation of its current input – each gate can choose non-deterministically whether to delay or not. As a result, the number of possible transitions from a given state can be as high as 2^n, where n is the number of gates. This sometimes (but not always) makes it more expensive to represent the transition relation.

As a second example of processes, the following program uses a variable `semaphore` to implement mutual exclusion between two asynchronous processes. Each process has four states: `idle`, `entering`, `critical` and `exiting`. The entering state indicates that the process wants to enter its critical region. If the variable `semaphore` is zero, it goes to the `critical` state, and sets `semaphore` to one. On exiting its critical region, the process sets `semaphore` to zero again.

```
MODULE main
VAR
  semaphore : boolean;
  proc1 : process user;
  proc2 : process user;
ASSIGN
  init(semaphore) := 0;
SPEC
  AG !(proc1.state = critical & proc2.state = critical)

MODULE user
VAR
  state : {idle,entering ,critical,exiting};
ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle,entering};
      state = entering & !semaphore : critical;
      state = critical : {critical,exiting};
      state = exiting : idle;
      1 : state;
    esac;
  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting : 0;
      1 : semaphore;
```

```
    esac;
FAIRNESS
  running
```

If any of the specification is false, the SMV model checker attempts to produce a counterexample, proving that the specification is false. This is not always possible, since formulas preceded by existential path quantifiers cannot be proved false by showing a single execution path. Similarly, subformulas preceded by universal path quantifier cannot be proved true by showing a single execution path. In addition, some formulas require infinite execution paths as counterexamples. In this case, the model checker outputs a looping path up to and including the first repetition of a state.

In the case of the semaphore program, suppose that the specification were changed to

```
  AG (proc1.state = entering -> AF proc1.state = critical)
```

In other words, we specify that if `proc1` wants to enter its critical region, it eventually does. The output of the model checker in this case is shown in Figure 1. The counterexample shows a path with `proc1` going to the `entering` state, followed by a loop in which `proc2` repeatedly enters its critical region and returns to its idle state, with `proc1` only executing only while `proc2` is in its critical region. This path shows that the specification is false, since proc1 never enters its critical region. Note that in the printout of an execution sequence, only the values of variables that change are printed, to make it easier to follow the action in systems with a large number of variables.

Although the parallel assignment mechanism should be suitable to most purposes, it is possible in SMV to specify the transition relation directly as a propositional formula in terms of the current and next values of the state variables. Any current/next state pair is in the transition relation if and only if the value of the formula is one. Similarly, it is possible to give the set of possible initial states as a formula in terms of only the current state variables. These two functions are accomplished by the TRANS and INIT statements respectively. As an example, here is a description the three inverter ring using only TRANS and INIT:

```
MODULE main
VAR
  gate1 : inverter(gate3.output);
  gate2 : inverter(gate1.output);
  gate3 : inverter(gate2.output);
SPEC
  (AG AF gate1.out)& (AG AF !gate1.out)

MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = 0
TRANS
  next(output) = !input | next(output) = output
```

```
specification is false

AG (proc1.state = entering -> AF proc1.s... is false:

.semaphore = 0
.proc1.state = idle
.proc2.state = idle

next state:

[executing process.proc1]

next state:

.proc1.state = entering

AF proc1.state = critical is false:

[executing process .proc2]

next state:

[executing process .proc2]
.proc2.state = entering

next state:

[executing process .proc1]
.semaphore = 1
.proc2.state = critical

next state:

[executing process .proc2]

next state:

[executing process .proc2]
.proc2.state = exiting

next state:

.semaphore = 0
.proc2.state = idle
```

Figure 1: Model checker output for semaphore example.

According to the TRANS declaration, for each inverter, the next value of the output is equal either to the negation of the input, or to the current value of the output. Thus, in effect, each gate can choose non-deterministically whether or not to delay.

Similarly, one can use the INVAR declaration to specify *invariants* that every state in the transition system must satisfy, which results in restricting the transition relation to only those states. The INVAR clause corresponds to the *normal* assignments. For example,

```
ASSIGN
  x := y + 1;
```

can be written as

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

The use of TRANS, INVAR and INIT is not recommended, since logical absurdities in these declarations can lead to unimplementable descriptions. For example, one could declare the logical constant 0 to represent the transition relation, resulting in a system with no transitions at all. However, the flexibility of these mechanisms may be useful for those writing translators from other languages to SMV.

To summarize, the SMV language is designed to be flexible in terms of the styles of models it can describe. It is possible to fairly concisely describe synchronous or asynchronous systems, to describe detailed deterministic models or abstract nondeterministic models, and to exploit the modular structure of a system to make the description more concise. It is also possible to write logical absurdities if one desires to, and also sometimes if one does not desire to, using the TRANS and INIT declarations. By using only the parallel assignment mechanism, however, this problem can be avoided. The language is designed to exploit the capabilities of the symbolic model checking technique. As a result the available data types are all static and finite. No attempt has been made to support a particular model of communication between concurrent processes. In addition, there is no explicit support for some features of communicating process models such as sequential composition. Since the full generality of the symbolic model checking technique is available through the SMV language, it is possible that translators from various languages, process models, and intermediate formats could be created. In particular, existing silicon compilers could be used to translate high level languages with rich feature sets into a low level form (such as a Mealy machine) that could be readily translated into the SMV language.

2 Syntax and Semantics

This section describes the syntax and semantics of the SMV input language in detail.

2.1 Lexical conventions

An atom in the syntax described below may be any sequence of characters in the set {A-Z,a-z,0-9,_,-}, beginning with an alphabetic character. All characters in a name are significant, and case is significant. Whitespace characters are space, tab and newline. Any string starting with two dashes ("--") and ending with a new line is a comment. A `number` is any sequence of digits. Any other tokens recognized by the parser are enclosed in quotes in the syntax expressions below.

2.2 Expressions

Expressions are constructed from variables, constants, and a collection of operators, including Boolean connectives, integer arithmetic operators, and case expressions. The syntax of expressions is as follows.

```
expr ::
        atom                    ;; a symbolic constant
        | number                ;; a numeric constant
        | id                    ;; a variable identifier
        | "!" expr              ;; logical not
        | expr1 "&" expr2       ;; logical and
        | expr1 "|" expr2       ;; logical or
        | expr1 "->" expr2      ;; logical implication
        | expr1 "<->" expr2     ;; logical equivalence
        | expr1 "=" expr2       ;; equality
        | expr1 "!=" expr2      ;; disequality
        | expr1 "<" expr2       ;; less than
        | expr1 ">" expr2       ;; greater than
        | expr1 "<=" expr2      ;; less that or equal
        | expr1 ">=" expr2      ;; greater than or equal
        | expr1 "+" expr2       ;; integer addition
        | expr1 "-" expr2       ;; integer subtraction
        | expr1 "*" expr2       ;; integer multiplication
        | expr1 "/" expr2       ;; integer division
        | expr1 "mod" expr2     ;; integer remainder
        | "next" "(" id ")"     ;; next value
        | set_expr              ;; a set expression
        | case_expr             ;; a case expression
```

An `id`, or identifier, is a symbol or expression which identifies an object, such as a variable or defined symbol. Since an `id` can be an atom, there is a possible ambiguity if a variable or defined symbol has the same name as a symbolic constant. Such an ambiguity is flagged by the interpreter as an error. The expression `next(x)` refers to the value of identifier `x` in next state (see section 2.5). The order of parsing precedence from high to low is

```
*,/
+,-
mod
=,!=,<,>,<=,>=
!
&
```

```
        |
        ->,<->
```

Operators of equal precedence associate to the left, except for the implication `->`, which associates to the right. Parentheses may be used to group expressions.

A `case` expression has the syntax

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

A case expression returns the value of the first expression on the right hand side, such that the corresponding condition on the left hand side is true. Thus, if `expr_a1` is true, then the result is `expr_b1`. Otherwise, if `expr_a2` is true, then the result is `expr_b2`, *etc.* If none of the expressions on the left hand side is true, the result of the `case` expression is the numeric value 1. It is an error for any expression on the left hand side to return a value other than the truth values 0 or 1.

A set expression has the syntax

```
set_expr ::
        "{" val1 "," ... "," valn "}"
        | expr1 "in" expr2      ;; set inclusion predicate
        | expr1 "union" expr2   ;; set union
```

A set can be defined by enumerating its elements inside curly braces. The elements of the set can be numbers or symbolic constants. The inclusion operator tests a value for membership in a set. The union operator takes the union of two sets. If either argument is a number or symbolic value instead of a set, it is coerced to a singleton set.

2.3 State variables

A state of the model is an assignment of values to a set of state variables. These variables (and also instances of modules) are declared by the notation

```
decl :: "VAR"
          atom1 ":" type1 ";"
          atom2 ":" type2 ";"
          ...
```

The type associated with a variable declaration can be either a Boolean, a scalar, a user defined module, or an array of any of these (including arrays of arrays). A type specifier has the syntax

```
type :: boolean
        | "{" val1 "," val2 "," ... valn "}"
        | "array" expr1 ".." expr2 "of" type
        | atom [ "(" expr1 "," expr2 "," ... exprn ")" ]
        | "process" atom [ "(" expr1 "," expr2 "," ... exprn ")" ]

val  :: atom | number
```

A variable of type `boolean` can take on the numerical values 0 and 1 (representing false and true, respectively). In the case of a list of values enclosed in quotes (where atoms are taken to be symbolic constants), the variable is a scalar which take any these values. In the case of an `array` declaration, the expression `expr1` is the lower bound on the subscript, and the expression `expr2` is the upper bound. Both of these expressions must evaluate to integer constants. Finally, an atom optionally followed by a list of expressions in parentheses indicates an instance of module atom (see section 2.10). The keyword `process` causes the module to be instantiated as an asynchronous process (see 2.13).

2.4 The `ASSIGN` declaration

An assignment declaration has the form

```
decl :: "ASSIGN"
           dest1 ":=" expr1 ";"
           dest2 ":=" expr2 ";"
           ...

dest :: atom
        | "init" "(" atom ")"
        | "next" "(" atom ")"
```

On the left hand side of the assignment, `atom` denotes the current value of a variable, `init(atom)` denotes its initial value, and `next(atom)` denotes its value in the next state. If the expression on the right hand side evaluates to an integer or symbolic constant, the assignment simply means that the left hand side is equal to the right hand side. On the other hand, if the expression evaluates to a set, then the assignment means that the left hand side is contained in that set. It is an error if the value of the expression is not contained in the range of the variable on the left hand side.

In order for a program to be implementable, there must be some order in which the assignments can be executed such that no variable is assigned after its value is referenced. This is not the case if there is a circular dependency among the assignments in any given process. Hence, such a condition is an error. In addition, it is an error for a variable to be assigned a value more than once at any given time. To be precise, it is an error if:

1. the next or current value of a variable is assigned more than once in a given process, or

2. the initial value of a variable is assigned more than once in the program, or

3. the current value and the initial value of a variable are both assigned in the program, or
4. the current value and the next value of a variable are both assigned in the program

2.5 The `TRANS` declaration

The transition relation R of the model is a set of current state/next state pairs. Whether or not a given pair is in this set is determined by a Boolean valued expression T, introduced by the `TRANS` keyword. The syntax of a `TRANS` declaration is

```
decl :: "TRANS" expr
```

It is an error for the expression to yield any value other than 0 or 1. If there is more than one `TRANS` declaration, the transition relation is the conjunction of all of `TRANS` declarations.

2.6 The `INIT` declaration

The set of initial states of the model is determined by a Boolean expression under the `INIT` keyword. The syntax of an `INIT` declaration is

```
decl :: "INIT" expr
```

It is an error for the expression to contain the `next()` operator, or to yield any value other than 0 or 1. If there is more than one `INIT` declaration, the initial set is the conjunction of all of the `INIT` declarations.

2.7 The `INVAR` declaration

The set of all states of the model is restricted to those that satisfy a Boolean expression under the `INVAR` keyword. Thus, `INVAR` defines an *invariant* on the transition system. The syntax of an `INVAR` declaration is

```
decl :: "INVAR" expr
```

As in the case of `INIT`, it is an error for the expression to contain the `next()` operator, or to yield any value other than 0 or 1. If there is more than one `INVAR` declaration, the invariant is the conjunction of all of the `INVAR` declarations.

2.8 The `SPEC` declaration

The system specification is given as a formula in the temporal logic CTL, introduced by the keyword `SPEC`. The syntax of this declaration is

```
decl :: "SPEC" ctlform
```

A CTL formula has the syntax

```
ctlform ::
    expr ;; a Boolean expression
    | "!" ctlform ;; logical not
    | ctlform1 "&" ctlform2     ;; logical and
    | ctlform1 "|" ctlform2     ;; logical or
    | ctlform1 "->" ctlform2    ;; logical implies
    | ctlform1 "<->" ctlform2   ;; logical equivalence
    | "E" pathform          ;; existential path quantifier
    | "A" pathform          ;; universal path quantifier
```

The syntax of a path formula is

```
pathform ::
        "X" ctlform                     ;; next time
        "F" ctlform                     ;; eventually
        "G" ctlform                     ;; globally
        ctlform1 "U" ctlform2           ;; until
```

The order of precedence of operators is (from high to low)

```
        E,A,X,F,G,U
        !
        &
        |
        ->,<->
```

Operators of equal precedence associate to the left, except for the implication `->`, which associates to the right. Parentheses may be used to group expressions. It is an error for an expression in a CTL formula to contain a `next()` operator or to return a value other than 0 or 1. If there is more than one `SPEC` declaration, the specification is the conjunction of all of the `SPEC` declarations. However, each of the `SPEC` formulas is evaluated and the results are reported separately, one by one, in the order of the `SPEC` declations in the program text.

2.9 The `FAIRNESS` declaration

A fairness constraint is a CTL formula which is assumed to be true infinitely often in all fair execution paths. When evaluating specifications, the model checker considers path quantifiers to apply only to fair paths. Fairness constraints are declared using the following syntax:

```
decl:: "FAIRNESS" ctlform
```

A path is considered fair if and only if all fairness constraints declared in this manner are true infinitely often.

2.10 The PRINT declaration

Sometimes it is desired to find out which states satisfy a particular specification, rather than checking whether all of the reachable states satisfy it. The `PRINT` declaration evaluates a specification and prints a formula describing the set of *reachable* states that satisfy this formula. In particular,

```
PRINT 1
```

prints a formula describing the set of all reachable states.

```
decl:: "PRINT" ctlform
     | "PRINT" header ":" ctlform
```

where the `header` tells SMV which variables should appear in the formula:

```
header:: "hide" id1 "," id2 "," ... idn
       | "expose" id1 "," id2 "," ... idn
```

For example,

```
PRINT expose x, y: x = y | y = z
```

will print a formula describing all possible values of `x` and `y` variables in all the reachable states satisfying the formula `x = y | y = z`. If the `expose` keyword is changed to `hide`, then the formula will contain all of the state variables except `x` and `y`.

2.11 The DEFINE declaration

In order to make descriptions more concise, a symbol can be associated with a commonly used expression. The syntax for this declaration is

```
decl :: "DEFINE"
          atom1 ":=" expr1 ";"
          atom2 ":=" expr2 ";"
          ...
          atomn ":=" expr3 ";"
```

When every identifier referring to the symbol on the left hand side occurs in an expression, it is replaced by the expression on the right hand side. The expression on the right hand side is always evaluated in its original context, however (see the next section for an explanation of contexts). Forward references to defined symbols are allowed, but circular definitions are not allowed, and result in an error.

2.12 Modules

A module is an encapsulated collection of declarations. Once defined, a module can be reused as many times as necessary. Modules can also be parameterized, so that each instance of a module can refer to different data values. A module can contain instances of other modules, allowing a structural hierarchy to be built. The syntax of a module is as follows.

```
module ::
        "MODULE" atom [ "(" atom1 "," atom2 "," ... atomn ")" ]
          decl1
          decl2
          ...
          decl3
```

The atom immediately following the keyword "MODULE" is the name associated with the module. Module names are drawn from a separate name space from other names in the program, and hence may clash with names of variables and definitions. The optional list of atoms in parentheses are the formal parameters of the module. Whenever these parameters occur in expressions within the module, they are replaced by the actual parameters which are supplied when the module is instantiated (see below).

An *instance* of a module is created using the VAR declaration (see section 2.3) This declaration supplies a name for the instance, and also a list of actual parameters, which are assigned to the formal parameters in the module definition. An actual parameter can be any legal expression. It is an error is the number of actual parameters is different from the number of formal parameters. The semantics of module instantiation is similar to call-by-reference. For example, in the following program fragment:

```
...
VAR
  a : boolean;
  b : foo(a);
...
MODULE foo(x)
ASSIGN
  x := 1;
```

the variable `b` is assigned the value `1`. This distinguishes the call-by-reference mechanism from a call-by-value scheme. Now consider the following program:

```
...
DEFINE
  a := 0;
VAR
  b : bar(a);
...
MODULE bar(x)
DEFINE
```

```
  a := 1;
  y := x;
```

In this program, the value of `y` is 0. On the other hand, using a call-by-name mechanism, the value of `y` would be 1, since `a` would be substituted as an expression for `x`.

Forward references to module names are allowed, but circular references are not, and result in an error.

2.13 Identifiers

An `id`, or identifier, is an expression which references an object. Objects are instances of modules, variables, and defined symbols. The syntax of an identifier is as follows.

```
id ::
        atom
        | id "." atom
        | id "[" expr "]"
```

An *atom* identifies the object of that name as defined in a `VAR` or `DEFINE` declaration. If a identifies an instance of a module, then the expression $a.b$ identifies the component object named b of instance a. This is precisely analogous to accessing a component of a structured data type. Note that an actual parameter of module instance a can identify another module instance b, allowing a to access components of b, as in the following example:

```
...
VAR
  a : foo(b);
  b : bar(a);
...
MODULE foo(x)
DEFINE
  c := x.p | x.q;

MODULE bar(x)
VAR
  p : boolean;
  q : boolean;
```

Here, the value of `c` is the logical or of `p` and `q`.

If a identifies an array, the expression $a[b]$ identifies element b of array a. It is an error for the expression b to evaluate to a number outside the subscript bounds of array a, or to a symbolic value.

2.14 The main module

The syntax of an SMV program is

```
program ::
        module1
        module2
        ...
        modulen
```

There must be one module with the name `main` and no formal parameters. The module `main` is the one evaluated by the interpreter.

2.15 Processes

Processes are used to model interleaving concurrency. A *process* is a module which is instantiated using the keyword `process` (see section 2.3). The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Each instance of a process has a special Boolean variable associated with it called `running`. The value of this variable is 1 if and only if the process instance is currently selected for execution.

3 Examples

In this section, we look at the performance of the SMV symbolic model checker for two hardware examples – a synchronous fair bus arbiter, and an asynchronous distributed mutual exclusion ring circuit (the one studied by David Dill in his thesis [Dil89] and designed by Alain Martin [Mar85]).

3.1 Synchronous arbiter

The synchronous arbiter circuit is an example of a synchronous finite state machine. It is composed of a "daisy chain" of arbiter cells depicted in Figure 2. Under normal operation, the arbiter grants the bus on each clock cycle to the requester with the highest priority. Each arbiter cell receives a "bus grant" input from the next higher priority cell. If this signal is true, and the cell's "request" input is true, then the cell activates its "acknowledge" output, and negates "bus grant" to the next lower priority cell. On the other hand, if the "request" input is false, then the "bus grant" input is passed along to the next cell via the "bus grant" output. Despite this priority scheme, the bus arbiter is designed to insure that every requester eventually is granted the bus. During light bus traffic, the priority scheme is used, but as the bus approaches saturation, the arbiter reverts to a round-robin scheme. This is accomplished by means of a "token", which is passed in a cyclic manner from the first cell down to the last, and then back to the first. The "token" moves once each clock cycle. When the "token" passes a cell whose "request" is active, it sets a flag "waiting". The "waiting" flag remains set as long as the request persists. When the token returns to

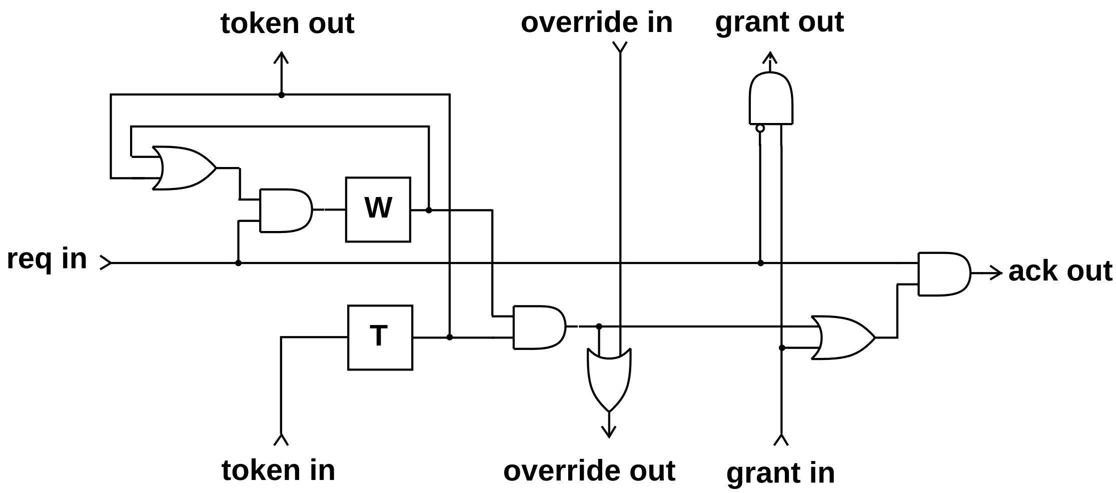


Figure 2: Cell of synchronous arbiter circuit.

that cell, if the "waiting" flag is still set, the cell receives immediate highest priority. This is accomplished by asserting an output called "override". This signal propagates to the highest priority cell and negates its "bus grant" input.

The specifications for the arbiter circuit are as follows:

1. No two acknowledge outputs are asserted simultaneously.

2. Every persistent request is eventually acknowledged.

3. Acknowledge is not asserted without request.

Expressed in CTL, they are:

1. $\forall i \neq j : \mathbf{AG}\ \neg(ack_i \wedge ack_j)$

2. $\forall i : \mathbf{AG}\ \mathbf{AF}\ (req_i \Rightarrow ack_i)$

3. $\forall i : \mathbf{AG}\ (ack_i \Rightarrow req_i)$

The quantifiers are bounded to range over the finite set of cells, so these quantified formulas can be expanded into finite CTL formulas. Figure 3 gives the SMV description of a five cell arbiter and its specification.

To run the symbolic model checker on this example, we use the command

```
smv -f syncarb.smv
```

The option `-f` indicates that a forward search of the state space of the model should be made before checking the specifications. This technique will be dealt with shortly.

Figure 4 plots the performance of the symbolic model checking procedure for this example in terms of several measures. First, the size of the transition relation in OBDD nodes. Second, the total run time (on a Sun3, running an implementation in the C language), and third, the maximum number of OBDD nodes used at any given time. The latter number should be regarded as being accurate only to within a factor of two, since the garbage collector in the implementation scavenges for unreferenced nodes only when the number of nodes doubles. We observe that as the number of cells in the circuit increases, the size of the transition relation increases linearly. The execution time is well fit by a quadratic curve.

To obtain polynomial performance for this example, it was necessary to add a wrinkle to the symbolic model checking algorithm (the `-f` option. It is often the case that circuits are "well behaved" in the part of their state space which is reachable from the initial state, but not otherwise. In the case of the synchronous arbiter, only states with one token in the ring are reachable. However, the symbolic model checking technique considers all states, including states with multiple tokens. This becomes a problem when we consider the highest priority cell, which is granted the bus by default when no other requesters override. If we compute the set of states in which this cell necessarily grants the bus in k steps, we obtain the set in which, for every waiting cell i, there is no token at cell $i-k \bmod n$ (hence a token does not reach cell i in k steps). Unfortunately, this is not a set which can be compactly represented as an OBDD. This is analogous to the problem of representing a shifter circuit using OBDDs – there is no variable ordering which produces a compact OBDD for all shift distances k. As a result, the time required to compute $AFack_0$ is exponential, roughly doubling with each added cell.

On the other hand, if we first compute the set of reachable states, and then restrict the evaluation of the temporal operators to that set, the result is unchanged, but the verification time becomes polynomial. When we restrict to states with only one token, we only have to represent the set of states where cell $i+k \bmod n$ is not waiting, where i is the position of the single token.

3.2 Asynchronous state machines

An asynchronous finite state machine can be viewed as a collection of parallel processes whose actions are interleaved arbitrarily. This allows us to make an important optimization in the symbolic model checking technique: we observe that the set of states reachable by one step of the system is the union of the sets of states reachable by one step of each individual process. Using this fact, we can avoid computing the transition relation of the system and instead use only the transition relations of the individual processes.

Our example of an asynchronous state machine is the distributed mutual exclusion (DME) circuit of Alain Martin [Mar85]. It is a speed-independent circuit and makes use of special two-way mutual exclusion circuits as components. Figure 5 is a diagram of a single cell of the distributed mutual-exclusion ring (DME). The circuit works by passing a token around the ring, via the request and acknowledge signals RR and RA. A user of the DME gains exclusive access to the resource via the request and acknowledge signals UR and UA.

The specifications of the DME circuit are as follows:

1. No two users are acknowledged simultaneously.

2. An acknowledgment is not output without a request.

3. An acknowledgment is not removed while a request persists.

```
MODULE arbiter-element(above,below,init-token)
VAR
  Persistent : boolean;
  Token : boolean;
  Request : boolean;
ASSIGN
  init(Token) := init-token;
  next(Token) := token-in;
  init(Persistent) := 0;
  next(Persistent) := Request & (Persistent | Token);
DEFINE
  above.token-in := Token;
  override-out := above.override-out | (Persistent & Token);
  grant-out := !Request & below.grant-out;
  ack-out := Request & (Persistent & Token | below.grant-out);
SPEC
  AG ((ack-out -> Request) & AF (!Request | ack-out))

MODULE main
VAR
  e5 : arbiter-element(self,e4,0);
  e4 : arbiter-element(e5,e3,0);
  e3 : arbiter-element(e4,e2,0);
  e2 : arbiter-element(e3,e1,0);
  e1 : arbiter-element(e2,self,1);
DEFINE
  grant-in := 1;
  e1.token-in := token-in;
  override-out := 0;
  grant-out := grant-in & !e1.override-out;
SPEC
  AG (
      !(e1.ack-out & e2.ack-out)

      & !(e1.ack-out & e3.ack-out)
      & !(e2.ack-out & e3.ack-out)

      & !(e1.ack-out & e4.ack-out)
      & !(e2.ack-out & e4.ack-out)
      & !(e3.ack-out & e4.ack-out)

      & !(e1.ack-out & e5.ack-out)
      & !(e2.ack-out & e5.ack-out)
      & !(e3.ack-out & e5.ack-out)
      & !(e4.ack-out & e5.ack-out)
     )
```

Figure 3: SMV program for synchronous arbiter example.

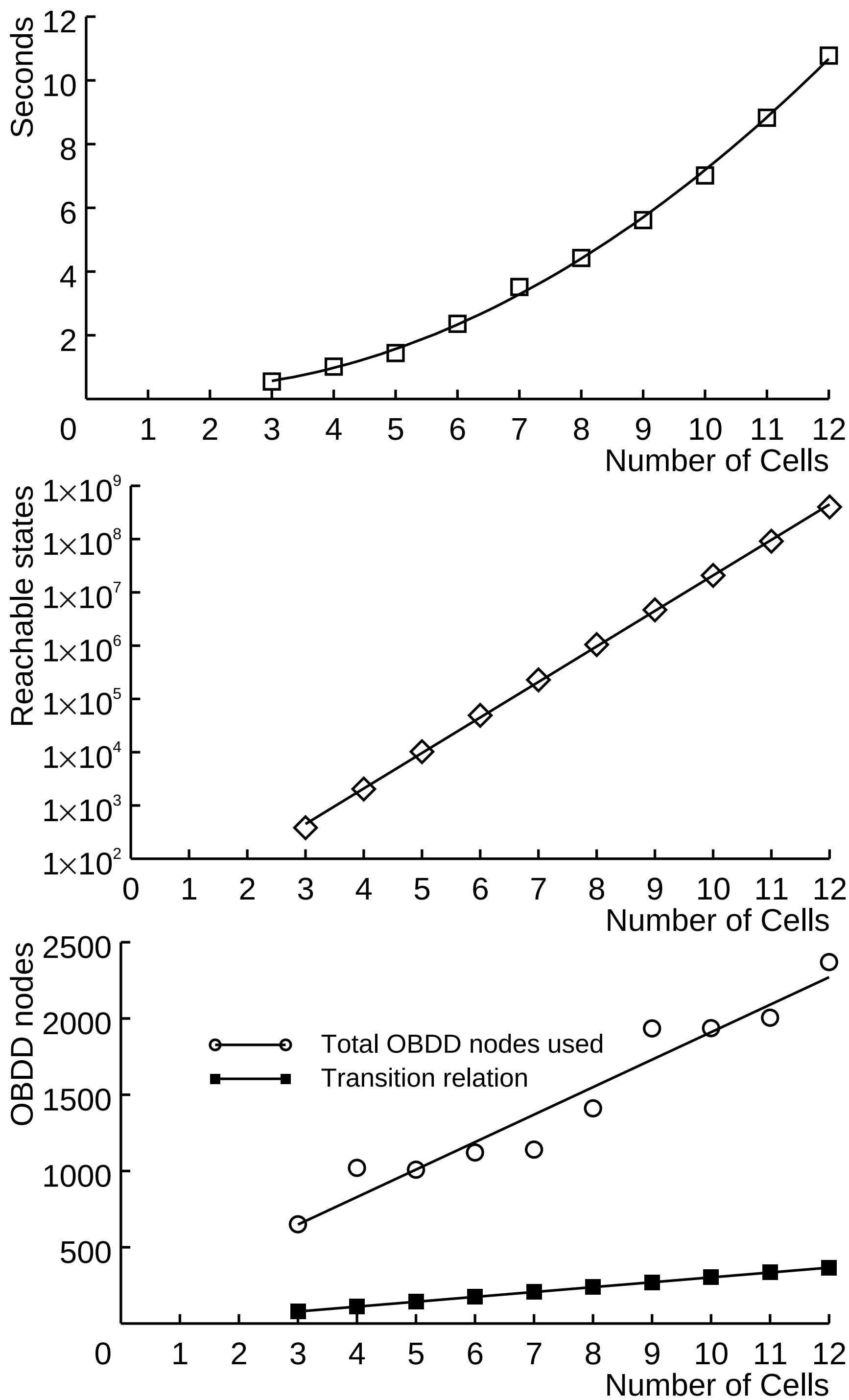


Figure 4: Performance – synchronous arbiter example.

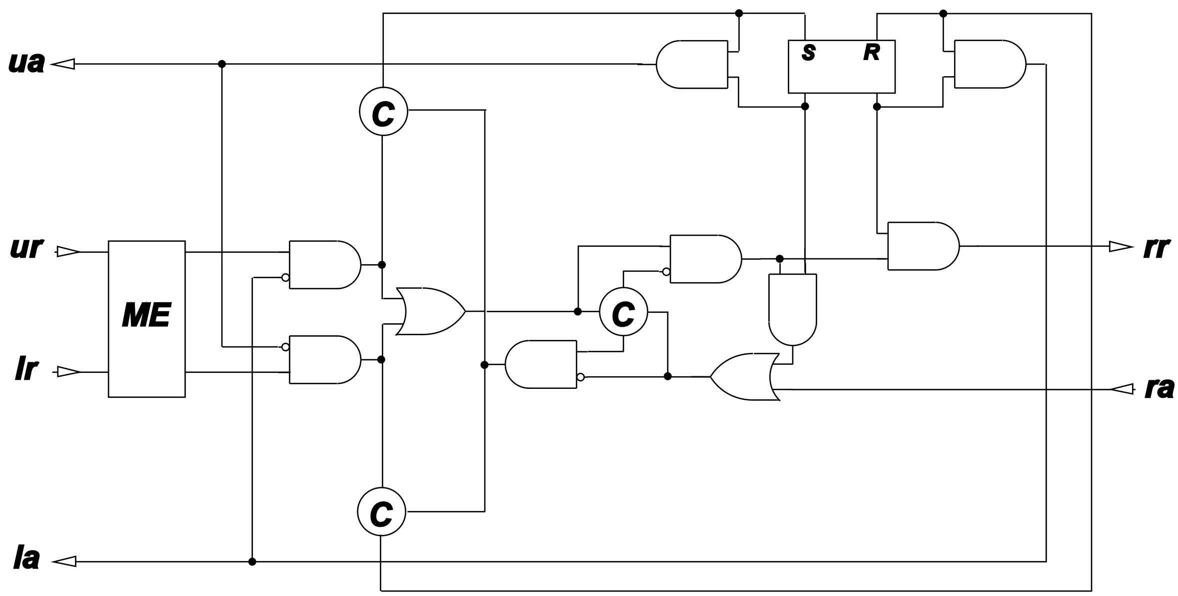


Figure 5: One cell of the DME circuit.

4. All requests are eventually acknowledged.

We will consider only the first specification, regarding mutual exclusion. The others are easily formulated in CTL, although the last requires the use of fairness constraints (see section 2.8) to guarantee that all gate delays are finite. The formalization of the mutual exclusion specification is

$$\forall 0 \leq i, j < n, i \neq j : \mathbf{AG}\ \neg(ack_i \wedge ack_j)$$

We examine the performance of the symbolic model checking algorithm in verifying this specification using three differing approaches. In method 1, we use a single process to model the entire system. Arbitrary delay of the gates is introduced by allowing each gate to choose non-deterministically whether to reevaluate its state or remain unchanged. The SMV description of this model is given in Figure 6. In method 2, we model each DME cell by a separate process (since there are 18 gates per cell, making a separate process for each gate is prohibitive). In method 3, we use the same model as in method 1, but reevaluate the the transition relation at each step of the forward search, restricting the evaluation to those transitions beginning in a state on the search frontier. This results in a sequence of approximations to the transition relation which are substantially more compact than the complete transition relation, at the expense of many reevaluations of the transition relation. This method of calculation is invoked by using the `-i` option to the model checker. The OBDD function *Restrict* of Coudert, Madre and Berthet is used to restrict the transition relation. In all three methods, we use the `-f` option to restrict the computation to the reachable states, since the state space of this circuit is quite sparse.

The performance curves for the three methods are shown in Figure 7. The disjunctive transition relation method requires $O(n^4)$ time, while the two conjunctive methods – with

```
MODULE  and-gate(in1,in2)
VAR     out : boolean;
ASSIGN  init(out) := 0;  next(out) := (in1 & in2) union out;

MODULE  and-gate-init(in1,in2,init-out)
VAR     out : boolean;
ASSIGN  init(out) := init-out;  next(out) := (in1 & in2) union out;

MODULE  or-gate(in1,in2)
VAR     out : boolean;
ASSIGN  init(out) := 0;  next(out) := (in1 | in2) union out;

MODULE  c-element(in1,in2)
VAR     out : boolean;
ASSIGN  init(out) := 0;
        next(out) := case in1 = in2 : in1 union out;  1 : out; esac;

MODULE  mutex-half(inp,other-out)
VAR     out : boolean;
ASSIGN  init(out) := 0;  next(out) := inp union out;
TRANS   !(next(out) & next(other-out))

MODULE  user
VAR     req : boolean;
ASSIGN  init(req) := 0;  next(req) := (!ack) union req;

MODULE  cell(left,right,token)
VAR     q : and-gate(f.out,n.out);    f : c-element(d.out,i.out);
        d : and-gate(b.out,!u.ack);   b : mutex-half(left.req,a.out);
        i : and-gate(h.out,!j.out);   h : c-element(g.out,j.out);
        n : and-gate-init(!e.out,!m.out,!token);  u : user;
        a : mutex-half(u.req,b.out);  c : and-gate(a.out,!left.ack);
        g : or-gate(c.out,d.out);     e : c-element(c.out,i.out);
        k : and-gate(g.out,!h.out);   l : and-gate(k.out,m.out);
        p : and-gate(k.out,n.out);    m : and-gate-init(!f.out,!n.out,token);
        r : and-gate(e.out,m.out);    j : or-gate(l.out,ack);
DEFINE  req := p.out;  left.ack := q.out;   u.ack := r.out;

MODULE  main
VAR     e-3 : cell(e-1,e-2,1);  e-2 : cell(e-3,e-1,0);  e-1 : cell(e-2,e-3,0);
SPEC  AG (
      !(e-1.u.ack & e-2.u.ack)
      & !(e-1.u.ack & e-3.u.ack)
      & !(e-2.u.ack & e-3.u.ack)
     )
```

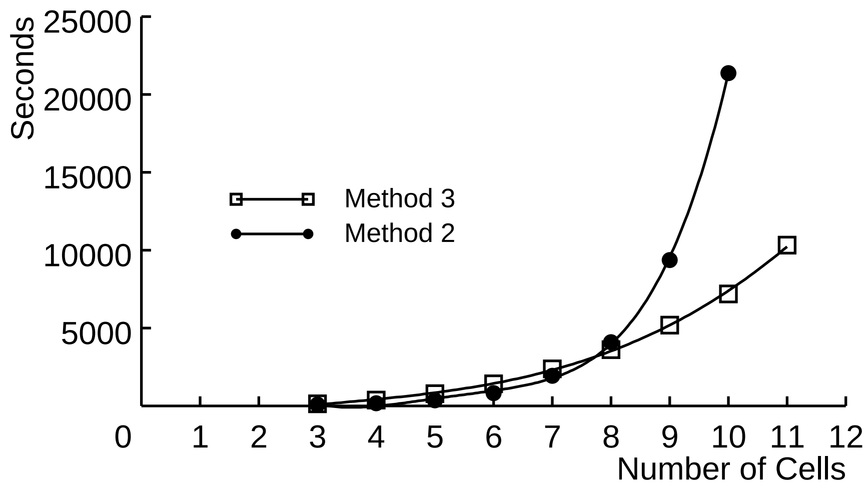
Figure 6: SMV program for DME circuit example.

unrestricted and restricted transition relation – require $O(n^3)$ time. As a result, the restricted transition relation method overtakes the disjunctive method at about 8 cells. At this point, the disadvantage of having to evaluate the transition relation at each step is outweighed by the better asymptotic performance. The difference in asymptotic performance can be explained by observing the growth in the OBDDs representing state sets in the forward search. The size of the largest such OBDDs as a function of the number of cells is plotted in Figure 7(c). As mentioned previously, the correlation between the number of steps taken by each process can make the representation of the reached state less efficient. Thus, the OBDD size for the reached state set runs linearly for methods 1 and 3, but quadratically for method 2. The overall time complexity of $O(n^3)$ for methods 1 and 3 derives from three factors: a linear increase in the transition relation OBDD, a linear increase in the state set OBDD, and a linear increase in the number of iterations. For method 2, the quadratic increase in the state set OBDD results in an overall $O(n^4)$ time complexity. Note that the number of reachable states increases roughly a factor of ten with each added cell (see Figure 8).

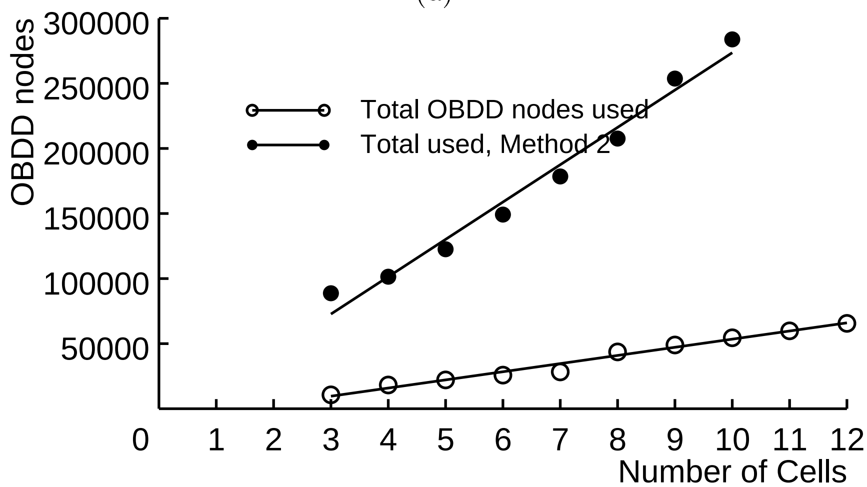
References

[Dil89] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.

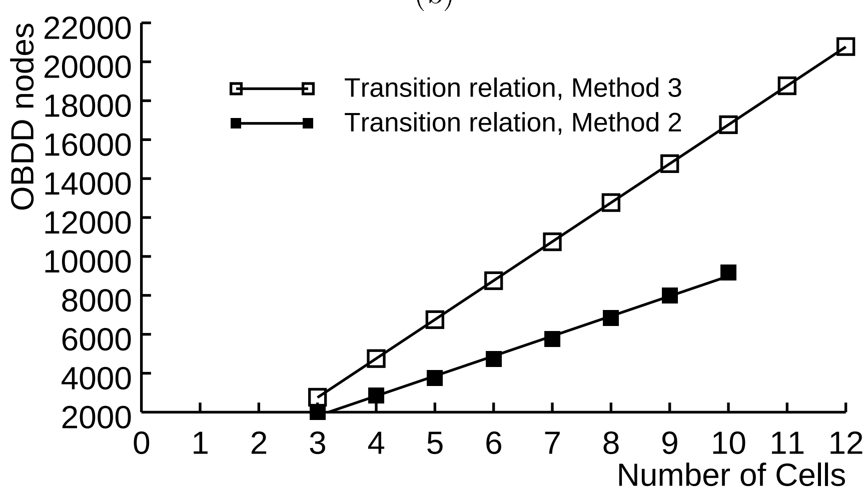
[Mar85] A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, 1985.



(a)

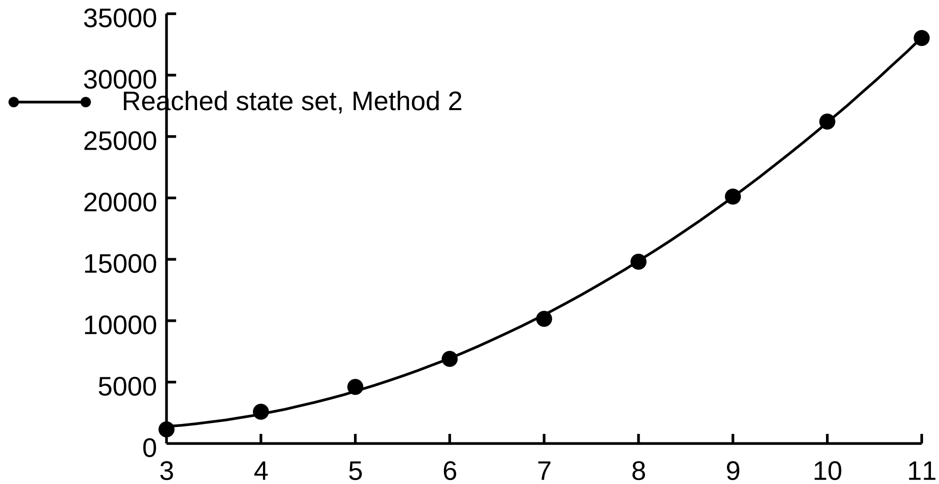


(b)

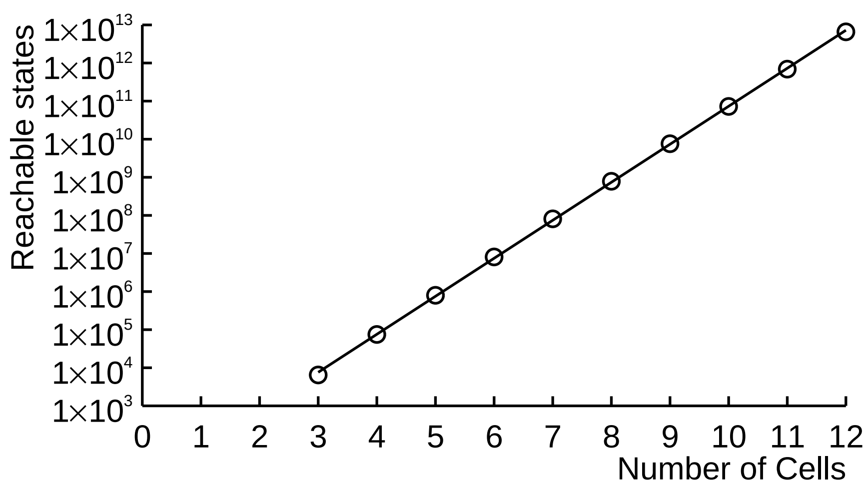


(c)

Figure 7: Performance for DME circuit example.



(a)



(b)

Figure 8: State set size for DME circuit example.