Chapter 3

The SMV system

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 basic data types in the language are finite scalar types. 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 provide a symbolic description of the transition relation of a finite Kripke structure. Any propositional formula 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. A program can be viewed as a system of simultaneous equations, whose solutions determine the next state. By checking programs for multiple assignments to the same variable, circular dependencies, and type errors, the compiler 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 the possible applications of symbolic model checking to hardware verification.

3.1 An informal introduction

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 compiler using a collection of Boolean variables, so that the transition relation may be 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 (a system of simultaneous equations), 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 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, carryin 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, however, since they don't require introducing a new variable into the OBDD 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 your 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 compiler detects both multiple assignments and circular dependencies, 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 the data dependencies. The same logic applies to defined symbols. As a result, all legal SMV programs are realizable.

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 arbitrarily in the execution sequence of the program. This is useful for describing communication protocols, asynchronous circuits, or other systems whose actions are not synchronized (including synchronous circuits with more than one clock). 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. The specification of this program states that the output of `gate1` 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 execute every process infinitely often, hence the output of a given gate may remain constant, regardless of changes 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 those execution paths along which a given CTL formula is true infinitely often. 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 interleaving 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 the combine the results (cf. section 2.4.2). This can yield a substantial savings in space in representing the transition relation.

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 union operator allows us to express a nondeterministic choice between two expressions. 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. The relative advantages of interleaving and simultaneous models of asynchronous systems are discussed in section 2.4.2.

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 specification in the program 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 a showing a single execution path. Similarly, subformulas preceded by universal path quantifier cannot be proved true by a 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 3.1. 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 the 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 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 of 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)
```

```
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 3.1: Model checker output for semaphore example

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

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 nondeterministically whether or not to delay. The use of TRANS 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 (false) 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 (*eg.*, synchronous or asynchronous message passing). 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.

3.2 The input language

This section describes the various constructs of the SMV input language, and their syntax.

3.2.1 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 newline 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.

3.2.2 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       ;; 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 compiler as an error. The expression `next(x)` refers to the value of identifier `x` in the next state (see section 3.2.3). The order of parsing precedence from high to low is

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

Operators of equal precedence associate to the left. Parentheses may be used to group expressions.

A `case` expression has the syntax

```
case_expr ::
        "case"
          expr_a1 ":" expr_b1 ";"
          expr_a2 ":" expr_b2 ";"
          ...
        "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 "," val2 "," ... "}"
        | 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.

3.2.3 Declarations

The `VAR` declaration

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 Boolean, scalar, or a user defined module. A type specifier has the syntax

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

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 set brackets (where atoms are taken to be symbolic constants), the variable is a scalar which can take any of these values. Finally, an `atom` optionally followed by a list of expressions in parentheses indicates an instance of module `atom` (cf. section 3.2.4). The keyword `process` causes the module to be instantiated as an asynchronous process (cf. section 3.2.6).

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 more than once simultaneously. 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, or

5. there is a circular dependency, or

6. the current value of a variable depends on the next value of a variable.

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, 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.

The `INIT` declaration

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

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. 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.

The FAIR 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 :: "FAIR" ctlform
```

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

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 ";"
          ...
```

When every an identifier referring to the symbol on the left hand side occurs in an expression, it is replaced by the *value* of the expression on the right hand side (not the expression itself). Forward references to defined symbols are allowed, but circular definitions are not allowed, and result in an error.

3.2.4 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 ::
        [ "OPAQUE" ]
        "MODULE" atom [ "(" atom1 "," atom2 "," ... ")" ]
          decl1
          decl2
          ...
```

The optional keyword OPAQUE is explained in the section on identifiers. 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.

A *instance* of a module is created using the VAR declaration (cf. section 3.2.3). 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 if 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, consider the following program fragment:

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

The variable a is assigned the value 1. 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 assigned to y is 0. Using a call-by-name (macro expansion) 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.

3.2.5 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
```

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 the keyword OPAQUE appears before a module definition, then the variables of an instance of that module are not externally accessible. Thus, the following program fragment is not legal:

```
...
VAR
  a : foo();
DEFINE
  b := a.x;
...
OPAQUE MODULE foo()
VAR
  x : boolean;
...
```

3.2.6 Processes

Processes are used to model interleaving concurrency, with shared variables. A *process* is a module which is instantiated using the keyword `process` (cf. section 3.2.3). The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel, simultaneously. Each instance of a process has special variable Boolean associated with it called `running`. The value of this variable is `1` if and only if the process instance is currently selected for execution. The rule for determining whether a given variable is allowed to change value when a given process is executing is as follows: if the next value of a given variable is not assigned in the currently executing process, but is assigned in some other process, then the next value is the same as the current value.

3.2.7 Programs

The syntax of an SMV program is

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

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

3.3 Formal semantics

In this section we assign a formal semantics to SMV programs. In essence, a program is viewed as a system of equations whose solutions determine the transition relation and initial states of a Kripke structure. In fact, this semantics assigns meaning to some programs which are not actually accepted by the compiler due to the rules regarding multiple assignments and circular dependencies. Here, we define a semantics for a subset of the language which does not include the `process` keyword. This subset will be called SMV.0. The semantics of SMV.0 is syntax directed – the denotation of a program is a function of the denotations of its syntactic components. It is also *compositional* with regard to bisimulation and simulation, as we will prove in chapter 5. This makes it possible to use compositional proof methods for verifying SMV.0 programs, including induction over the structure of programs. The semantics for SMV.1, which includes the `process` keyword, is given in appendix A.

3.3.1 The model

The set N of *names*, is the set of all character strings made up of the letters, the digits, the underscore and the minus sign characters, beginning with a letter. The *store* $L = L_V \cup L_H$ is made up of two disjoint, countably infinite sets of *locations* L_V and L_H. We will call the former the *visible* locations, and the latter the *hidden* locations. The set of locations L is defined recursively. It is the least set such that

1. if $n \in N$, then $n \in L_V$, and
2. if $l \in L_V$ and $n \in N$, then $l.n \in L_V$, and
3. if $l \in L_V$, then $.l \in L_H$.

The set of values V is the union of the integers in the range $[-2^{31}, 2^{31}-1]$ and N, the set of names. A *state* $x : L \to V$ is a function from locations to values. Let $S = L \to V$ be the set of all possible states.

If p is a declaration, then its denotation $[\![p]\!]$ is a triple (T, I, R). The T component is a partial function from L to the finite subsets of V.

If l is a location, then $T(l)$, when defined, is the *type* of l – the set of values that can be assigned to location l. The component $I \subseteq S$ is the set of initial states. Finally, the component $R \subseteq S \times S$ is the transition relation.

In the following sections, we define the denotations of the various kinds of declarations. We then define a composition operator $\|$ which gives the denotation of a program in terms of its declarations.

3.3.2 Expressions

An expression denotes a function from states to finite subsets of V, according to the following rules:

1. If v is a value, then $[\![v]\!](x) = \{v\}$.

2. If l is a location, then $[\![l]\!](x) = \{x(l)\}$.

3. If e_1, e_2 are expressions, and o is one of

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

   then

   $$[\![e_1 \; o \; e_2]\!](x) = \{[\![o]\!](v_1, v_2) \mid v_1 \in [\![e_1]\!](x),\, v_2 \in [\![e_2]\!](x)\}$$

4. If e is an expression, then

   $$[\![!e]\!](x) = \{[\![!]\!](v) \mid v \in [\![e]\!](x)\}$$

5. If e_1, e_2 are expressions,

   $$[\![e_1 \; \texttt{union} \; e_2]\!](x) = [\![e_1]\!] \cup [\![e_2]\!]$$

6. If e_1, e_2 are expressions,

   $$[\![e_1 \; \texttt{in} \; e_2]\!](x) = [\![e_1]\!] \subseteq [\![e_2]\!]$$

The functions denoted by `+`, `-`, `*`, `/` are the usual functions of arithmetic modulo 2^{32}. The function denoted by `mod` is the positive remainder of division mod 2^{32}. The functions denoted by the relational operators `>`, `>=`, `<` and `<=` return 0 when the relation is false and 1 when the relation is true, and are defined for numeric values only. For non-numeric values, they return $\perp$. The equality operator `=` is defined for all values, and returns 0 when they are unequal, and 1 when they are equal. The functions denoted by the Boolean operators are `&` (for and), `|` (for or), `!` (for not), `->` (for implies) and `<->` (for logical equivalence) are defined only for the values 0 and 1, and return $\perp$ otherwise.

3.3.3 Assignments and definitions

There is no semantic difference between assignments and definitions. If l is a location, and e is an expression, then the assignment $l := e;$ denotes a triple (T, I, R), where

1. $T = \emptyset$
2. $I = S$
3. $R = \{(x, y) \in S^2 \mid l(x) \in [\![e]\!](x)\}$

The assignment `next`$(l) := e;$ denotes a triple (T, I, R) where

1. $T = \emptyset$
2. $I = S$
3. $R = \{(x, y) \in S^2 \mid l(y) \in [\![e]\!](x)\}$

The assignment `init`$(l) := e;$ denotes a triple (T, I, R) where

1. $T = \emptyset$
2. $I = \{x \in S \mid l(x) \in [\![e]\!](x)\}$
3. $R = S^2$

3.3.4 Variable declarations

If l is an identifier and $v_1, v_2, \ldots, v_n$ are values, then

$$\texttt{VAR}\ l\ :\ \{v_1, v_2, \ldots, v_n\};$$

denotes a triple (T, I, R) where

1. $T = \{(l, \{v_1, v_2, \ldots, v_n\})\}$
2. $I = \{x \in S \mid x(l) \in \{v_1, v_2, \ldots, v_n\})\}$
3. $R = \{(x, y) \in S \mid x(l), y(l) \in \{v_1, v_2, \ldots, v_n\})\}$

3.3.5 Renaming

Let $\phi : L \to L$ be a function from locations to locations. This in turn induces a map Φ on states, such that for all states x and locations l,

$$\Phi(x)(l) = x(\phi(l)).$$

If $M = (T, I, R)$, then let $\phi(M) = (T', I', R')$ where

1. $T'(\phi(l)) = T(l)$,
2. $I' = \{x \mid \Phi(x) \in I\}$ and
3. $R' = \{(x, y) \mid (\Phi(x), \Phi(y)) \in R\}$.

Note that the definition of T does not make sense if ϕ maps two locations with different types onto the same location. In this case, $\phi(M)$ is a type error. There are two rules regarding the renaming function ϕ which must be respected to allow compositional reasoning about SMV programs. These are:

1. A hidden location cannot be renamed to a visible location, and
2. Two distinct locations cannot be mapped to the same hidden location.

These rules are respected by the SMV.0 semantics. Notice that it is allowable to rename visible locations to hidden locations. In this way, we can accomplish both hiding and renaming with the same operator.

3.3.6 Parallel composition

The parallel composition of two processes $M_1 \parallel M_2$ is formed in two steps. First, a renaming is applied to map the hidden locations of M_1 and M_2 onto disjoint spaces. Then the union of the type functions T and the intersections of the initial sets I and the transition relations R are taken. Clearly, this does not make sense if the T components do not agree on the type of some location, since the union would not be a function. Formally, let $M_1 = (T_1, I_1, R_1)$ and $M_2 = (T_2, I_2, R_2)$. Let n_1 and n_2 be two distinct names. For $i \in 1, 2$, let $\phi_i(l) = .n_i.l$ for all $l \in L_H$ and $\phi_i(l) = l$ otherwise, and let $M'_i = \phi(M_i)$. The parallel composition $M = M_1 \parallel M_2$ is defined as follows:

1. $T = T'_1 \cup T'_2$
2. $I = I'_1 \cap I'_2$
3. $R = R'_1 \cap R'_2$

If $d_1, d_2, \ldots, d_k$ are declarations, then $[\![d_1 \;\; d_2 \;\; \ldots \;\; d_k]\!]$ is the parallel composition

$$[\![d_1]\!] \parallel [\![d_2]\!] \parallel \cdots \parallel [\![d_k]\!]$$

.

3.3.7 Instantiation

Suppose that module A is defined as follows:

$$\texttt{MODULE}\ A(n_1, n_2, \ldots, n_k)\ D$$

where $n_1, n_2, \ldots, n_k$ are distinct names and D is a sequence of declarations. Let $r, l_1, l_2, \ldots, l_k$ be visible locations. Let ϕ be a renaming, such that, for all $l \in L_V$,

1. for all $1 \leq i \leq k$: $\phi(n_i) = l_i$, and $\phi(n_i.l) = l_i.l$,
2. for all $n \in N - \{n_1, n_2, \ldots, n_k\}$, $\phi(n) = r.n$, and $\phi(n.l) = r.n.l$,
3. $\phi(.l) = .l$

Then $[\![\texttt{VAR}\ r\ :\ A(l_1, l_2, \ldots, l_k);]\!] = \phi(D)$.

On the other hand, suppose that A is defined as follows:

$$\texttt{OPAQUE MODULE}\ A(n_1, n_2, \ldots, n_k)\ D$$

where $n_1, n_2, \ldots, n_k$ are distinct names and D is a sequence of declarations. Let $r, l_1, l_2, \ldots, l_k$ be visible locations. Let m_1 and m_2 be distinct names in N. Let ϕ be a renaming, such that, for all $l \in L_V$,

1. for all $1 \leq i \leq k$: $\phi(n_i) = l_i$, and $\phi(n_i.l) = l_i.l$,
2. for all $n \in N - \{n_1, n_2, \ldots, n_k\}$, $\phi(n) = .m_1.n$, and $\phi(n.l) = .m_1.n.l$,
3. $\phi(.l) = .m_2.l$

Then $[\![\texttt{VAR}\ r\ :\ A(l_1, l_2, \ldots, l_k);]\!] = \phi(D)$.

3.3.8 Specifications

Each program is associated with a Kripke structure which determines the truth value of CTL formulas in the specification. The atomic propositions in this case are all the Boolean valued expressions. The Kripke structure associated with a program whose denotation is the triple (T, I, R) is a Kripke model $K = (S, R, L')$ where

1. S is the set of states defined above,
2. R is the transition relation, and
3. if e is an expression, then

$$L'(e) = \{x \in S \mid [\![e]\!](x) = \{1\}\}$$

The specification is a formula f in CTL with fairness constraints. It is satisfied exactly when $K, s_0 \models f$ for all $s_0 \in I$.