VERDS Modeling Language

Wenhui Zhang

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences

P.O.Box 8718, Beijing 100190, China

2012-11-26

This document explains the syntax of the *VERDS* modeling language (VML), the language of the models (i.e., hierarchical discrete systems) for the verification tool *verds*. A model specified in VML is called a *VERDS* verification model (VVM).

An atom may be any sequence of characters starting with a character in the set {a-z_} and followed by a possibly empty sequence of characters belonging to the set {a-z0-9_}. A number is any sequence of digits. A digit belongs to the set {0-9}.

All characters and case in a name are significant. Whitespace characters are space (SPACE), tab (TAB) and newline (RET). Comments in *verds* language is any string starting with two slashes (`//') and ending with a newline. Any other tokens recognized by the parser are enclosed in quotes in the syntax expressions below. Grammar productions enclosed in square brackets (`[]') are optional.

Keywords in VML include

VVM	MAIN	DEFINE	VAR	INIT
TRANS	FAIRNESS	SPEC	ASSUMPTION	GUARANTEE
PROC	MODULE	PROCEDURE	RETURN	int
char	running	pid	for	in

Logical operators include

X	F	G	U	R
A	E	&	|	!

The empty string is denoted by ε. For simplicity, we define the operator listof(x) and parlistof(x), that may be used in various situations for the parameter x.

listof(x) :: ε | x ";" listof(x)

parlistof(x) :: x | x "," parlistof(x)

1. Expressions

Expressions are constructed from variables, constants, and a collection of operators, including temporal operators, Boolean connectives, and integer arithmetic operators.

1.1. Numeric Expressions

Numeric expressions are expressions built only from current state variables. The syntax of numeric expressions is as follows:

```
numeric_expr ::
          symb_const                        ;; a symbolic constant
        | numb_const                        ;; a numeric constant
        | char_const                        ;; a numeric constant
        | variable_id                       ;; a variable identifier
        | "P" "(" variable_id ")"           ;; a marked variable id
        | "(" numeric_expr ")"
        | numeric_expr "+" numeric_expr     ;; integer addition
        | numeric_expr "-" numeric_expr     ;; integer subtraction
        | numeric_expr "*" numeric_expr     ;; integer multiplication
        | numeric_expr "/" numeric_expr     ;; integer division
        | numeric_expr "%" numeric_expr     ;; integer remainder

symb_const ::    atom

numb_const ::    number

char_const ::    'a' | … | 'z'

                 | 'A' | … | 'Z'

                 | '0' | … | '9'

                 | '\n'

variable_id ::   atom | atom"."atom | variable_id "[" numeric _expr "]"
```

A symbolic constant is represented by an atom, which are predefined ones, including the atom "pid". Such constants are meant to be used in particular sections of the declarations.

A numeric constant is represented by a number. A simple variable identifier is either an atom representing a simple variable or such a variable identifier preceded by a process identifier (represented by the first atom in the construct atom"."atom) in order to refer a local variable of a process. A simple variable identifier followed by a sequence of indices represents an element of an array variable.

The order of parsing precedence for operators from high to low is:

*, /, %
+, -

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

The marked variable identifiers are only appropriate in specifications of guarantees.

1.2. Logic Expressions

Logic expressions are expressions built from numeric expressions. Logic expressions can be used to specify sets of states, e.g. the initial set of states. The syntax of logic expressions is as follows:

```
logic_expr ::
          "TRUE"                              ;; The boolean constant 1
        | "FALSE"                             ;; The boolean constant 0
        | logic_const                         ;; predefined constants
        | "(" logic _expr ")"
        | numeric_expr "=" numeric_expr       ;; equality
        | numeric_expr "!=" numeric_expr      ;; inequality
        | numeric_expr "<" numeric_expr       ;; less than
        | numeric_expr ">" numeric_expr       ;; greater than
        | numeric_expr "<=" numeric_expr      ;; less than or equal
        | numeric_expr ">=" numeric_expr      ;; greater than or equal
        | logic_expr "&" logic_expr           ;; logical and
        | logic_expr "|" logic_expr           ;; logical or
        | "!" logic_expr                      ;; logical not
        | "for" atom "in" range":" logic_expr  ;; for all statement

logic_const :: atom

range :: number ".." number | "{" atom "," atom "," ... "," atom "}"
```

A logical constant is represented by an atom, which are predefined ones, including the atom “running”. Such constants are meant to be used in particular sections of the declarations.

The order of parsing precedence for operators from high to low is:

=, !=, <, >, <=, >=
!
&
|

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

1.3. Temporal Logic Expressions

Temporal logic expressions are expressions built from logic expressions. Logic expressions can be used to specify properties of the finite state machine. The syntax of temporal logic expressions is as follows:

```
temporal_expr ::
          logic_expr
        | "AX" temporal_expr
        | "AG" temporal_expr
        | "AF" temporal_expr
        | "A" "(" temporal_expr "R" temporal_expr ")"
        | "A" "(" temporal_expr "U" temporal_expr ")"
        | "EX" temporal_expr
        | "EG" temporal_expr
        | "EF" temporal_expr
        | "E" "(" temporal_expr "R" temporal_expr ")"
        | "E" "(" temporal_expr "U" temporal_expr ")"
        | temporal_expr "&" temporal_expr              ;; logical and
        | temporal_expr "|" temporal_expr              ;; logical or
        | "!" temporal_expr                            ;; logical not
```

1.4. Assignment Expressions

Assignment expressions are built from variable identifiers and logical expressions. The syntax of assignment expressions is as follows:

```
assignment ::
```

```
        "(" variable_id "," ... variable_id ")"
        ":="
        "(" ext_numeric_expr "," ... ext_numeric_expr ")"

ext_numeric_expr :: numeric_expr | "*"
```

Extended numeric expressions augment numeric expressions with “*”. The meaning of this constant is “any value” of some given type in the context.

```
varname :: atom ["[]"]

param :: varname | number

ext_assignment :: [procedure_call "&"] assignment ["&" logic_expr]

procedure_call :: atom "(" [ parlistof(param) ] ")"  //**//
```

2. Hierarchical Discrete Systems

A hierarchical discrete system consists three parts: the variables representing the state of the system, the initial states, and the transition relation. In addition, property specifications may be attached to a hierarchical discrete system, in order to check whether the system is correct with respect to the properties. The transition relation may be divided into different modules.

2.1. Types and State Variables

There are integers, characters, bounded numbers and enumerative types and record types. These types are declared by the notation:

```
type :: range
        | "int"
        | "char"
        | "record" "{" var_declaration "}"
```

Two enumeration types must either be identical or with disjoint sets of elements.

A state of the model is an assignment of values to a set of state variables. These variables are declared by the notation:

variable :: atom | variable "[" number ".." number "]"

simple_var_declaration :: variable ":" type

var_declaration :: listof(simple_var_declaration)

A variable is either a simple variable or an array variable. The type associated with a variable declaration can be either a scalar or an enumeration of a set of atoms.

A variable of type Boolean may be represented by a variable of type 0..1 such that *x=1* means *x* for the Boolean variable *x*, *x=0* means *not x*, and negating the Boolean variable *x* may be implemented by the arithmetic expression *1-x*.

2.2. Initial Values of State Variables

Initial values of state variables are specified by a list of formulas constraining the state variables. As a special case, the initial values of state variables may be specified as a list of formulas of the form *x=c* where *x* is a variable identifier and *c* is a constant. The initial values are declared by the notation:

init_declaration :: listof(logic_expr)

2.3. Transition Relations

The transition relation of a process is implemented by a list of simple transition relations. The transition relation is declared by the notation:

simple_trans_relation :: logic_expr ":" ext_assignment ["&" "RETURN"]

trans_relation :: listof(simple_trans_relation)

2.4. Fairness Declarations

The transition relation may be augmented with fairness requirements that impose restrictions on the valid execution paths. The specification is declared by the notation:

fairness_declaration :: listof(logic_expr)

The logical constant “running” is a special proposition that may be used in a fairness declaration. The proposition is satisfied when the process in which the fairness is declared is executed.

2.5. Specification Declarations

The specification of system and process properties is represented by a temporal logic expression. The specification is declared by the notation:

spec_declaration :: listof(temporal_expr)

Procedure Declarations

A procedure is an encapsulated collection of declarations. Once defined, a procedure can be reused as many times as necessary.

The syntax of a procedure declaration is as follows.

```
procedure ::
        "PROCEDURE"     atom "(" [parlistof(varname)] ")"
        "VAR"           var_declaration
        "INIT"          init_declaration
        "TRANS"         trans_declaration
        [
        "ASSUMPTION"    [var_declaration] listof(logic_expr)
        "GUARANTEE"     listof(logic_expr)
        ]
```

The *atom* immediately following the keyword "PROCEDURE" is the name associated with the procedure. The optional par_list in parentheses is the list of the formal parameters of the procedure. The symbols "[]" indicates that the atom preceding the symbols represents an array variable. Whenever these parameters occur in expressions within the procedure, they are replaced by the actual parameters which are supplied when the procedure is instantiated.

2.6. Module Declarations

A module is an encapsulated collection of declarations. Once defined, a module can be reused as many times as necessary. Modules can also be so that each instance of a module can refer to different data values.

The syntax of a module declaration is as follows.

```
module ::
        "MODULE"        atom "(" [parlistof(varname)] ")"
        "VAR"           var_declaration
        "INIT"          init_declaration
        "TRANS"         trans_declaration
        [ "FAIRNESS"    fairness_declaration ]
```

The *atom* immediately following the keyword "MODULE" is the name associated with the module. The optional par_list in parentheses is the list of the formal parameters of the module. The symbols "[]" indicates that the atom preceding the symbols represents an array variable. Whenever these parameters occur in expressions within the module, they are replaced by the actual parameters which are supplied when the module is instantiated.

The symbolic constant "pid" is a special constant that may be used in a module declaration. The constant is interpreted as the numerical constant representing the id of the process (the first process has id=0), which is an instance of the module.

2.7. Multi-Process System Declarations

A hierarchical discrete system may have one or more processes that are instances of one or more module declarations. The syntax of a hierarchical discrete system (called a verds verification model) declaration is as follows.

```
verds_verification_model ::
        "VVM"           [ atom ]
        [ "DEFINE"      definition_declaration ]
        "VAR"           var_declaration
        "INIT"          init_declaration
        "PROC"          process_declaration
        [ "FAIRNESS"    fairness_declaration ]
        [ "SPEC"        spec_declaration ]

definition_declaration ::
        string_latin_letters "=" string_visible_characters "\n"
        ...
        string_latin_letters "=" string_visible_characters "\n"

simple_process_declaration :: atom ":" atom "("[parlistof(param)]")"
```

```
        | "for" atom "in" range":" atom ":" atom "("[parlistof(param)]")"

process_declaration :: listof(simple_process_declaration)
```

The optional atom after the keyword VVM is the name of the hierarchical discrete system.

A list of definitions may be provided. A defined word (which is a string of Latin letters) is an abbreviation of the defining string of visible characters, and the end-line symbol indicates the end of the definition. If a defining string is long, a backslash '\' may be used to ignore the end-line character immediately after this backslash.

A process is an instance of a module. The first atom in the process declaration is the name of the process (or the module instance), the second atom is the name of a module and the atoms in the optional list are parameters passed to the module.

The variables in the var_declaration section are global variables that can be used in all modules.

The complete verification model consists of a verds_verification_model declaration followed by a list of module declarations that are referred to in the process declaration.

2.8. Single Process System Declarations

For simplicity, a single process hierarchical discrete system may be declared by the following syntax as well.

```
single_process_verds_verification_model ::
        "VVM"           [ atom ]
        "VAR"           var_declaration
        "INIT"          init_declaration
        "TRANS"         trans_declaration
        [ "FAIRNESS"    fairness_declaration ]
        [ "SPEC"        spec_declaration ]
```

This declaration is similar to the previous declaration of a VVM with the difference that the process declaration is replaced by a transition declaration.