Rule-based Automatic Synthesizable VHDL Model Generation

Choon Kim
Intergraph Corporation
Huntsville, AL 35894-0001

Abstract

This paper presents a new approach to the automatic generation of a synthesizable VHDL model for the multiple synthesizable subsets environment. Our approach is based on the idea of independent and user-definable synthesis rule set. Unlike the previous methods, the proposed method separates a synthesis rule set from the model generator. The rules are consulted and used as the base for inferring the algorithm/construct of the model during the model generation phase. By changing or creating the independent rule set, a designer is able to generate a synthesizable VHDL model for different synthesizable subsets. In the implementation of the proposed method, a set of engineering parameters is used as input data for the circuit description.

1. Introduction

Automatic synthesizable VHDL model generation is an important issue among designers as synthesis-oriented design gains increasing support. It helps a designer save valuable design time as the automatic simulatable VHDL model generator does. In the synthesizable VHDL model generation, however, there are several problems to be carefully considered. The study presented in this paper is an approach to two problems associated with synthesizable VHDL model generation: synthesizable subset of VHDL and multiple synthesizable subsets.

The first problem stems from the fact that the synthesizable set of VHDL is not the full set of VHDL. Though the synthesizable subset covers most VHDL constructs and is continuously improving as synthesis technology advances, there are still certain VHDL constructs that are simulatable but are either ignored or unsupported by the synthesizer[1]. For the successful pass of the synthesis filter of the synthesizer, the VHDL model must include any unsupported VHDL construct. While ignored VHDL constructs may not cause trouble with the synthesis filter directly, the synthesizer may produce an incorrect circuit depending on the nature of the ignored construct. As a result, the simulation results (before and after synthesis) may not be consistent.

The second problem is caused by the lack of an industry-wide standard for a synthesizable VHDL subset at present time. It is true that most VHDL synthesizers currently available support very similar synthesizable subsets. However, it is noted that synthesizable subsets supported by those synthesizers are not identical[2]. Each synthesizer may provide some unique features and capabilities depending on its synthesis technology. The different synthesis technology makes different synthesizable subsets. From the point of view of the designer, the synthesizable subset of VHDL is a moving target depending on a specific synthesizer and a specific version. The synthesizable subset may change as either a different synthesizer is used or the version of a synthesizer is changed. It is necessary for a designer to know the details of the subset changes in order to make a VHDL model work with a changed synthesizable subset.

As more VHDL synthesizers are commercially available, it is often necessary to create a synthesizable VHDL model for multiple synthesizers (multiple synthesizable subsets). When a designer is in such an environment, current methods do not provide much help to a designer. One solution is to provide a designer a set of model generators for all the different synthesis subsets required. This solution may be too expensive and an intolerable burden to some designers, depending on the situation of their design environment.

2. Synthesis Subset of VHDL

As mentioned in the introduction, there is currently no industry-wide standard for the synthesizable subset of VHDL. The Figure 1 shows graphically the relationship between the full set of VHDL and different subsets supported by different synthesizers (the figure is not scaled).

![synthesizable VHDL subsets](image)

Figure 1. Synthesizable VHDL Subsets

The complement of the supported constructs are either ignored or unsupported. Many constructs are commonly supported while some constructs are supported by a particular synthesizer. There are also some constructs that no synthesizer is supporting.
List 1 shows some of the VHDL constructs that may not be fully supported by a synthesizer. Note that the list shows only a general idea about ignored/unsupported VHDL constructs, and the list may not be accurate depending on the synthesizer. Detailed information about these constructs is found in the specific synthesizer manual.

- generic in the entity declaration
- default value for port in the entity declaration
- entity statement - multiple architectures
- global signals
- configurations
- separate compilations
- default values for the parameter in subprogram
- resolution functions
- recursion in subprogram
- overloading
- physical data type
- floating data type
- multidimensional array data type
- access data type
- file data type
- incomplete data type
- register type signal
- bus type signal
- initial value of signal
- initial value of variable
- buffer type interface
- linkage type interface
- alias
- user-defined attributes(except synthesizer-specific ones)
- some of the predefined attributes
- disconnection
- some operators(with limiting conditions)
- allocator
- wait statement(with various conditions about location, number, etc)
- assertion
- guarded signal
- transfer mode signal
- after(i.e., time delay specification)
- while loop
- no-iteration scheme loop
- guarded block
- port in block
- generic in block
- sensitivity list of process
- multiple waveform specification
- severity level type
- time type
- now function
- textio-related constructs

When a model generator produces a synthesizable model, the contents of List 1 need to be carefully considered. In general, there exist various workarounds for some unsupported/ignored constructs. There are also desirable modeling styles recommended for the successful synthesis process. Some of such knowledge has been used to develop workarounds within the model generator.

3. User-defined Synthesis Rule Set

3.1 Analysis of Synthesis Rules

A synthesizer has a set of rules that a designer needs to follow for successful synthesizing. In general, the rules are explained in the synthesis manual. The contents of a rule are divided into two parts. The first part consists of various conditions that a rule has("if" part). The second part is a conclusion or judgement about the synthesizability("then" part). The 'if' part can contain many different conditions. In this paper, we analyze the 'if' part into five main factors: location, object, attribute, exception, and others. The 'location' shows a place in a model where a rule is applied. The 'object' shows a particular target that a rule is associated with. The 'attribute' is a specific property of 'object'. The 'except' handles an exceptional condition that a rule may have. The 'others' is reserved for a case that no other factors can handle.

In order to describe a rule more accurately, some of the main factors are further subdivided. The 'location' is further divided into four factors: region, statement, statement_set, statement_location. The 'region' is a basic block where a rule is applied. The 'statement' points to a specific statement within a 'region'. A rule may be associated with a specific statement or the complement set of the statement(i.e., other than the target statement). Such a complement set is described by the 'statement_set'. Some rules require a specific location of a statement(e.g., wait within a process). The 'statement_location' is reserved for that condition.

The 'object' and the 'attribute' have their subfactors such as the 'object_set' and the 'attribute_set'. The 'object_set' indicates a complement set or other set(e.g., number of dimensions allowed for multiple-index array). The 'attribute_set' shows a complement set. The analysis of a rule is graphically shown in Figure 2.

3.2 Rule Specification Format(RSF)

The synthesis rules are specified in an external file by using Rule Specification Format(RSF). A synthesis rule described in English needs to be encoded for rule abstraction. Based on the analysis of synthesis rule in 3.1, a relatively simple but efficient structure is used in this study. It is a record structure with multiple fields based on the analysis of a rule in 3.1(see Figure 2 for the structure of rule encoding)[3].

232
Currently, a synthesis rule is represented as a record structure which has eleven fields (the number of fields may be changed, possibly increased, in the future research if necessary). The eleven fields are: region, statement, statement_set, statement_location, object, object_set, attribute, attribute_set, exception, others, and synthesizability. Note that a field name is not related to the definition in VHDL [4], rather, it was defined for RSF purposes only. The fields were chosen to describe a specific information segment of a synthesis rule. The fields are optional, but at least two fields, including synthesizability, should be listed in a record to be meaningful. If a particular field is not listed in a record, it is assumed that the missing field is not relevant or critical to the synthesizability checking process.

Some synthesizers have a synthesis directive which can turn on or turn off the synthesizing process. Such a synthesis directive can be helpful to make a more flexible model (simulatable and synthesizable). To handle the synthesis directives, two special fields have been provided: synthesis_directive_on and synthesis_directive_off.

3.3 Field Values of RSF

Generally, pre-defined values are used for the field value. In a few cases, the user provides an arbitrary value (e.g., number of dimensions in array supported). The first four fields (region, statement, statement_set, and statement_location) represent information about the location of a domain where a rule is applied. The region field shows a location in a VHDL model. The values of the region fields are five VHDL library units and a context_clause. For example, 'entity_declaration' means the domain of rule is limited to the entity_declaration in a VHDL model.

The statement field represents a specific target VHDL statement to which a rule is applied. For example, a value of the statement field 'array', means a VHDL statement containing an array. The statement_set field has one value that is 'complement'. This field, if specified, indicates that the rule is applied to statements other than the target statement (i.e., complement set).

The statement_location field shows whether or not the target statement is at a particular location. For example, 'first' can be used to describe a rule such as "a wait statement, if used, must be a first statement within a process".

The object field represents a target object within a statement. Many values are available in the object field to describe a rule as accurately as possible. For example, 'access_type' points to an access type object in a statement. The object_set field, similar to the statement_set field, indicates that the rule is applied to the objects other than the target object (i.e., complement set), if specified. The object_set field has one value, 'complement'.

In general, the rules specified in a rule file are the unsupported/ignored ones because a supported rule does not need to be checked. The two objects, attribute and library name, are exceptions. In general, the user-defined attributes can be specified, but the use of them is unsupported. Instead, some synthesizers allow a set of predefined supported attributes. A user-defined library name is either supported or unsupported depending on a synthesizer. Therefore, only sup-
ported attributes and library names are specified in a rule file. The object field in a record needs a value that a user can enter. There are two object values that a user can enter. For the attributes, there is an object value, 'predef_attr_xxx' or 'userdef_attr_xxx' where xxx is a supported attribute name. For the library names, there is an object value, 'userdef_lib_yyy' where yyy is a logical name of a library. yyy can be 'all' if all user-defined libraries are supported.

The attribute field represents a particular property associated with a target object. For example, 'default_value' may be used as an attribute of an object (e.g., port) to describe a rule such as "Default values for ports are ignored". The attribute_set field, similar to the statement_set field or the object_set field, indicates that the rule is applied to attributes other than the target attribute (i.e., complement set), if specified.

The exception field is used to handle a situation where a particular type is unsupported except when it is used with user-defined attributes. For example, 'pre_defined_attribute' can be used to describe a rule such as "The use of floating-point type is unsupported except for floating-point constants used with synopsys-defined attributes". The others field is reserved to handle a situation that is not covered by any other field.

The synthesizability field shows a synthesizability of a rule and has four values to describe the nature of the synthesizability: ignored, supported, translated and unsupported. 'Ignored' is for a construct that may be necessary for simulation or documentation, but is ignored during synthesis. While it does not cause the failure of the synthesizing process, it still needs a designer's attention. For example, if a particular type is ignored when it is declared, the use of such a type may be prohibited and regarded as unsupported for synthesis. 'Supported' is for a construct that is synthesizable without any limitation. 'Translated' is for a construct that is ignored and is translated into another construct. It can be used to describe a rule such as "buffer and linkage are ignored and are translated to out and inout, respectively". 'Unsupported' is for a construct that may be necessary for simulation or documentation, but is unsupported during synthesis. An unsupported construct causes the failure of the synthesizing process, and needs to be fixed.

The value of two special fields (synthesis_directive_on and synthesis_directive_off) is a user-defined string that will be used as a synthesis directive. For example, "-- xxxx translate_off" can be a value of 'synthesis_directive_off' if the string is a directive which turns off the synthesizing process.

Some of the field values currently available

region = \{architecture_body, configuration_declaration, context_clause, entity_declaration, package_body, package_declaration\}

statement = \{array, assertion, attribute_specification, block, component_declaration, configuration_specification, declaration, delay_specification, entity_statement_part, generic_clause, guarded_signal_assignment, if_statement, port_clause, process, subprogram, type_declaration, use_clause, wait_on, wait_for, wait_until\}

statement_set = \{complement\}

statement_location = \{first, not_first\}

object = \{access_type, alias, allocator, constant, file_type, floating_point_type, generic, incomplete_type, integer_type, multiple_index, parameter, physical_type, port, pre-defined_attribute, predef_package_text, record_type, sensitivity_list, signal, use_of_access_type, use_of_file_type, use_of_floating_point_type, use_of_physical_type, use_of_record_type, predef_attr_xxx/userdef_attr_xxx\}

where, xxx is a supported attribute name, userdef_attr_xxx, userdef_lib_yyy, where, yyy = a user-defined supported library name

object_set = \{complement, deferred, N(lower-bound index)\}

attribute = \{default_value, initial_value, integer\}

attribute_set = \{complement\}

exception = \{user_defined_attribute\}

others = \{power_of_2, two\}

synthesizability = \{ignored, supported\}

synthesis_directive_on = \{user-defined command\}

synthesis_directive_off = \{user-defined command\}

Additional field values

In addition to the values listed above, several rules are directly associated with the VHDL reserved words. The examples include "bus/register declarations are unsupported", "alias is ignored", "Linkage is unsupported", etc. Therefore, the following values are selected for an efficient rule encoding. A value starts with 'vsw_' followed by a VHDL reserved word.

\{vsw_after, vsw_alias, vsw_bus, vsw_disconnect vsw_linkage, vsw_register, vsw_severity, vsw_transport, vsw_while\}

3.4 Syntax Definition of RSF in EBNF

Based on the definition of the record in RSF, the syntax for RSF is summarized as follows:

<table>
<thead>
<tr>
<th>RSF Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note: A comment starts with two adjacent hyphens and extends up to the end of the line.</td>
</tr>
</tbody>
</table>

Rule_Specification_File ::= synthesis_rule \{synthesis_rule\}
synthesis_rule ::= rule rule_number is \{' \rules_field_part \'}
[rule_number]\; |
rule_field_part ::= \{rule_field_item\}
rule_field_item ::= \{field_name \'} field_value \'; |
field_name ::= region | statement | statement_set | statement_location | object | object_set | attribute | attribute_set | exception | others | synthesis_directive_on | syn-
thesis_directive_off | synthesizability
field_value ::= { valid values listed in 3.3 }

3.5 RSF examples

rule 4 is
  -- "Default values for ports are ignored."
  { region: entity_declaration;
    statement: port_clauses;
    object: port;
    attribute: default_value;
    Synthesizability: ignored;
  } 4;

rule 17 is
  -- "Attribute, event, is supported as described with
  -- the wait statement"
  { statement: wait_statement;
    object: predef_attr_event;
    synthesizability: supported;
  } 17;

rule 33 is
  -- "the wait statement, if used, must be the first
  -- statement in a process"
  { statement: wait_statement;
    statement_location: not_first;
    synthesizability: unsupported;
  } 33;

4. Input Format for Automatic Model Generation

In this study, the user-specified engineering parameters are used as inputs for model generation. Several different methods have been reported in the past, such as graphical capture[5], PLD format[6], spreadsheet-like table[7], etc. Each method is effective in the area where the tool can best be applied. However, those methods require the designer to learn the particular input entry techniques.

Since it is common practice for an electronic designer to specify circuits using engineering parameters, this parameter-driven method allows an engineer to describe a VHDL model quickly and easily. This entry method allows the designers to use their familiar specifications, and virtually eliminates the need for learning the special entry scheme. The parameters are chosen to describe the circuit as detail as possible. The number and the nature of the parameters depend on the circuit.

An example is presented in List 2 & List 3. The example circuit is a 4-bit parallel-load, bidirectional-shift register. It is assumed that three control signals (the preset, the clear, and the initialize) are allowed. The operation is triggered at the rising_edge of the clock.

Circuit Type: Register
Name of the circuit: REG10
Number of bits: 4
Data input mode(s): PARALLEL,
Serial LEFT-SHIFT,
Serial RIGHT-SHIFT
Parallel input pin notation: NOBUS
Shift when SHLD is: 1
Shift left-to-right when RL is: 0
Bits tapped as Q outputs: ALL
Inhibit signal required: NO
Control signals: PRESET,
PRECLEAR,
INITIALIZE
Preset Register when PS is: 0
Preload Register when PC is: 0
Initialize Register when INIT: 0
Initialization value(HEX): 9
Control signal timing: SYNCHRONOUS
Static clock trigger: RISING_EDGE

List 2. List of User-specified Input Parameters

List 3 shows an example of simulatable VHDL output from the user-specified input parameters listed in List 2 (note that this example is not necessary a most optimized or efficient VHDL model for the circuit).

Note: 'X01Z' is used for the base type for this example.
-- VHDL model for 4-bit Parallel-loaded Bidirectional Shift
-- Register with Synchronous Preset, Preload and INIT
-- controls

-- Vendor-specific context_clause here...
Library ....
Use .......
entity REG10 is
generic(NBITS: integer := 4);
port( CK, PS, PC, INIT; SHLD, RL, I, RE: in X01Z;
P: in X01Z_vector(0 to NBITS-1);
Q: out X01Z_vector(0 to NBITS-1));
end REG10;
architecture BEHAVIOR of REG10 is
type VALUE_ARRAY is array(0 to 2, 0 to 3) of X01Z;
begin
REGISTER_PROCESS: process(CK)
variable REG_VALUE: X01Z_vector(NBITS-1 downto 0) := (others => 'X');
constant INITIAL_VALUES: VALUE_ARRAY :=
  ( others => '0', others => '1', '1', '0', '0', '1' );
variable INDEX: INTEGER := 0;
variable INVALID: INTEGER := 0;
begin
if rising_edge(CK) then -- vendor-specific rising-edge function
  -- check input

235
INVALID := 0;
if INIT /= '0' and INIT /= '1' then INVALID := 1;
elseif PS /= '0' and PS /= '1' then INVALID := 1;
elseif PC /= '0' and PC /= '1' then INVALID := 1;
elseif SHLD /= '0' and SHLD /= '1' then INVALID := 1;
elseif (RL /= '0' and RL /= '1') then INVALID := 1;
-- initialization
else PC = '0' then
  for I in 0 to NBITS-1 loop
    REG_VALUE(I) := INITIAL_VALUES(0,I);
  end loop;
else PS = '0' then
  for I in 0 to NBITS-1 loop
    REG_VALUE(I) := INITIAL_VALUES(1,I);
  end loop;
else INIT = '0' then
  for I in 0 to NBITS-1 loop
    REG_VALUE(I) := INITIAL_VALUES(2,I);
  end loop;
else -- actual shifting or loading process
if SHLD = '0' then -- parallel loading
  INDEX := 0;
else PARALLEL_LOAD: while INDEX < NBITS loop
  if PI(INDEX) /= '0' and PI(INDEX) /= '1' then
    INVALID := 1;
  exit PARALLEL_LOAD;
  elsif PI(INDEX) = '1' then
    REG_VALUE(INDEX) := '1';
  else REG_VALUE(INDEX) := '0';
  end if;
  INDEX := INDEX + 1;
end loop PARALLEL_LOAD;
else -- beginning of shifting part
if RL = '0' then -- shift right
  INDEX := 1;
while INDEX < NBITS loop
  REG_VALUE(INDEX-1) := REG_VALUE(INDEX);
  INDEX := INDEX + 1;
end loop;
REG_VALUE(NBITS-1) := LE;
else -- left-shift
  INDEX := NBITS-2;
while INDEX > -1 loop
  REG_VALUE(INDEX+1) := REG_VALUE(INDEX);
  INDEX := INDEX - 1;
end loop;
REG_VALUE(0) := RE;
end if; -- end of RL
end if; -- end of SHLD
end if; -- the end of check input
-- output assignment
if INVALID = 1 then
  Q <= (others => 'X');
  assert false report "Invalid input" severity error;
else
  for I in 0 to NBITS-1 loop
    if REG_VALUE(I) = '1' then Q(I) <= '1';
  else Q(I) <= '0';
  end if;
end loop;
end if;
end if; -- end of clock trigger
end process REGISTER_PROCESS;
end BEHAVIOR;

List 3. An Simulatable VHDL Model Example from List 2

5. Synthesizable Model Generation Process

5.1 Overall Flow

The flow diagram of the synthesizable model generation process is shown in Figure 3.

![Flow Diagram of Synthesizable Model Generation Process](image)

The first part of the process consists of stating the design specifications by providing the engineering parameters that describe the design. As mentioned before, the parameters are chosen to describe the circuit as detail as possible. After the user-defined design specifications (i.e., a set of engineering parameters) are entered, they are parsed and saved in the data base.

The second part of the process starts with a VHDL generator that reads and parses the rule set (rule abstraction). A synthesis filter is created in the memory as a result. The model generator then fetches the necessary information from the data base. Finally, the model generator writes a VHDL model following the built-in model generation guidelines. Each construct of the model is written after consulting with the synthesis filter about the synthesizability.

5.2 Rule Abstraction

The encoded rules in a rule file are parsed sequentially, in order, and stored in memory for synthesizability consulting. An incorrectly specified rule will be detected and causes a halt of the rule parsing process. On the completion of the parsing, a synthesis filter is dynamically created in memory. A synthesis filter is a memory data structure which contains all the encoded rules.
5.3 Following Model Generation Guidelines

During the model generation process, the model generator needs a set of guidelines that specify various conditions for the output VHDL model. Those guidelines are built-in (i.e., hard-coded) within the model generator. For example, the guidelines may include the information about the base type used, the library/package used, the logic value system used, the number of processes used, the existence of input validity checking, the algorithm used in the model, special synthesis conditions not covered by the rule set, etc. The model generation guidelines determine the overall shape and basic skeleton of the model.

5.4 Consulting with the synthesis filter

After successful creation of the synthesis filter, the model generator starts writing an output VHDL model with the set of user-specified input parameters. While the model generation guidelines determine the overall shape and the basic skeleton of the model, each construct of the model depends on the synthesis filter. Whenever a construct in the model is to be written, the model generator first consults with the synthesis filter in order to figure out whether or not the construct is supported.

Consulting is performed by invoking an interface function called 'consult_syn_rules(arguments)'. The arguments represent the fields of a compiled rule which specify the construct to be checked. The 'consult_syn_rules' function, when invoked with proper arguments, searches the synthesis filter to find a matching rule. If a matching rule is found, the function returns a rule number and the synthesizability information (i.e., supported, ignored, or unsupported) to the model generator. If no matching rule is found, the construct is assumed to be supported. The model writer takes an appropriate action (e.g., writing the construct or looking for a workaround, etc.) according to the result of consulting. The following algorithm describes the process.

```
start
pick up the first VHDL construct;
while ( VHDL construct to be written )
  ( if ( the construct does not need to be consulted about synthesizability ) write VHDL output; 
  else
    ( determine the arguments from the construct; 
      invoke the 'consult_syn_rules(arguments)' function; 
      case ( return_value ) 
        unsupported: ( call the 'Workaround' procedure; ) 
        ignored: 
          ( if ( the construct is simulation-specific and can be safely ignored ) 
            write VHDL output; 
          else
            call the 'Workaround' procedure; ) 
        others: -- supported case 
          write VHDL output; 
  end case
} pick up the next VHDL construct;
} finish
5.5. Workaround for unsupported constructs

As mentioned in Section 2, it is possible that some unsupported constructs can be rewritten by using the supported constructs (e.g., some type of "while-loop" can be replaced by "for-loop"). Also, some simulation-specific constructs can be safely ignored for synthesis (e.g., "assert-report-severity" construct). Some of this type of information has been used to develop workarounds within the model generator in this study (it is called 'Workaround' procedure).

If a construct is found as unsupported after consulting with the synthesis filter, the model generator starts looking for an alternative construct(s) that is logically identical to the unsupported construct. Such a construct, if found, may be used for the model after consulting with the synthesis filter again. For an example, suppose that the two-dimensional array is desired in the model, but, the consulting result indicates it is unsupported. The model generator has several options in this case, such as modeling without using two-dimensional array, or using an array of array structure if it is supported. One of the options will be used by the model generator to avoid the two-dimensional array construct.

6. Example

6.1 Register example

The Register model introduced on List 3 in Section 4 is used again as an example. First, the synthesizability of the Register model is checked. Basically, each construct in the Register model may be either supported, ignored, or unsupported depending on the synthesizer. In List 4 below, some well-known potential synthesis problems are marked with "****" for the example purpose. List 4 shows that, even in a simple model like this, writing a synthesizable model requires a designer to carefully consider various aspect of the limitations of the synthesizer.

-- Note: 'X01Z' is used for the base type for this example
-- VHDL model for 4-bit Parallel-loaded Bidirectional
-- Shift Register with Synchronous Preset, Precharge and
-- INIT controls

-- Vendor-specific context_clause here...
Library ....
Use ......
      **** Separate compilation
entity REG10 is
generic(NBITS: integer := 4);
      **** Generic
port( CK, PS, PC, INIT, RL, LE, RE: in X01Z;
    PI: in X01Z-vector(0 to NBITS-1);
Q: out X01Z_vector(0 to NBITS-1));
end REG10;

architecture BEHAVIOR of REG10 is

type VALUE_ARRAY is array(0 to 2, 0 to 3) of X01Z;

constant INITIAL_VALUES: VALUE_ARRAY :=
  (((others => '0'), (others => '1'), ('1', '0', '0', '1')));

variable INDEX: INTEGER := 0;  ***** Initial value of var.
variable INVALID: INTEGER := 0;  ***** Initial value of var.

begin
  REGISTER_PROCESS: process(CLK)
  begin
    variable REG_VALUE: X01Z_vector(NBITS-1 downto 0) :=
      (others => 'X');  ***** Initial value of var.
    if rising_edge(CLK) then  -- vendor-specific rising-edge trigger
      INVALID := 0;
      if INIT = '0' and INIT = '1' then INVALID := 1;
      elsif PS = '0' and PS = '1' then INVALID := 1;
      elsif PC = '0' and PC = '1' then INVALID := 1;
      elsif SHLD = '0' and SHLD = '1' then INVALID := 1;
      elsif (RL = '0' and RL = '1') then INVALID := 1;
      -- initialization
      if PC = '0' then
        for I in 0 to NBITS-1 loop
          REG_VALUE(I) := INITIAL_VALUES(I);  ***** Multidimensional array
        end loop;
      elsif PS = '0' then
        for I in 0 to NBITS-1 loop
          REG_VALUE(I) := INITIAL_VALUES(I);  ***** Multidimensional array
        end loop;
      elsif INIT = '0' then
        for I in 0 to NBITS-1 loop
          REG_VALUE(I) := INITIAL_VALUES(I);  ***** Multidimensional array
        end loop;
      else  -- actual shifting or loading process
        if SHLD = '0' then  -- parallel loading
          INDEX := 0;
          PARALLEL_LOAD: while INDEX < NBITS loop
            if PI(INDEX) = '0' and PI(INDEX) = '1' then
              INVALID := 1;
              exit PARALLEL_LOAD;
            elsif PI(INDEX) = '1' then
              REG_VALUE(INDEX) := '1';
            else
              REG_VALUE(INDEX) := '0';
            end if;
            INDEX := INDEX + 1;
          end loop PARALLEL_LOAD;
        else  -- beginning of shifting part
          if RL = '0' then  -- shift right
            INDEX := 1;
            while INDEX < NBITS loop
              REG_VALUE(INDEX-1) := REG_VALUE(INDEX);
              INDEX := INDEX + 1;
            end loop;
            REG_VALUE(NBITS-1) := LE;
          else  -- left-shift
            INDEX := NBITS-2;
            while INDEX > -1 loop
              REG_VALUE(INDEX+1) := REG_VALUE(INDEX);
              INDEX := INDEX - 1;
            end loop;
            REG_VALUE(0) := RE;
          end if;
          end of RL
          end if;
          end of SHLD
          end if;
          -- the end of check input
          -- output assignment
          if INVALID = 1 then
            Q := (others => 'X');
          assert false
          report "Invalid input" severity error;
          end assert
          else
            for I in 0 to NBITS-1 loop
              if REG_VALUE(I) = '1' then
                Q(I) := '1';
              else
                Q(I) := '0';
              end if;
            end loop;
            end if;
            end of clock trigger
            end process REGISTER_PROCESS;
            end BEHAVIOR;
          end if;
        end if;
      end if;
    end if;
  end process;
end BEHAVIOR;

List 4. Register Example Marked with Possible Synthesis Problems

6.2 Two Synthesizable Subsets and Results

As an example, two different synthesizable subsets were assumed in this section. Let's call them Subset A and Subset B. As shown on Figure 1 in Section 2, most parts of those two subsets are common, but there are some parts that they do not share. Some comparisons between two subsets for this Register example are as follows.

Construct: generic
Subset A: unsupported
Subset B: supported with limited conditions (e.g., only for integer)

Construct: multi-dimensional array
Subset A: unsupported, but, array of array is supported
Subset B: unsupported, but, array of array is supported

Construct: initial value of variable
Subset A: ignored
Subset B: unsupported

Construct: While-loop
Subset A: unsupported
Subset B: supported with limited conditions (e.g., there must be at least one wait statement in each enclosed logic branch)

Construct: Assert
Subset A: ignored
Subset B: ignored
Construct: Severity level type
Subset A: ignored
Subset B: unsupported

The following two VHDL outputs are results from the model generator. List 5 contains an output VHDL model based on the Subset A while List 6 contains an output VHDL model based on the Subset B. Note that all unsupported constructs were either removed, changed or rewritten with supported ones in both outputs.

-- Note: 'X01Z' is used for the base type for this example
-- VHDL model for 4-bit Parallel-loaded Bidirectional Shift
-- Register with Synchronous Preset, Preclear and INIT
-- controls

Library ...
Use ...

entity REG10 is
  port( CK, PS, PC, INIT, SHLD, RL, LE, RE: in X01Z;
       PI: in X01Z_vector(0 to 3);
       Q: out X01Z_vector(0 to 3));
end REG10;

architecture BEHAVIOR of REG10 is
  constant NBITS: POSITIVE := 4;
begin
  REGISTER_PROCESS: process
    variable REG_VALUE: X01Z_vector(NBITS-1 downto 0) :=
        (others => 'X');
    variable INVALID: INTEGER := 0;
  begin
    wait until CK'event and CK='1'; -- rising-edge trigger
    -- check input
    INVALID := 0;
    if INIT /= '0' and INIT /= '1' then INVALID := 1;
    elsif PS /= '0' and PS /= '1' then INVALID := 1;
    elsif PC /= '0' and PC /= '1' then INVALID := 1;
    elsif SHLD /= '0' and SHLD /= '1' then INVALID := 1;
    elsif (RL /= '0' and RL /= '1') then INVALID := 1;
    -- initialization
    elsif PC = '0' then REG_VALUE(0) := '0';
    REG_VALUE(1) := '0';
    REG_VALUE(2) := '0';
    REG_VALUE(3) := '0';
    elsif PS = '0' then REG_VALUE(0) := '1';
    REG_VALUE(1) := '1';
    REG_VALUE(2) := '1';
    REG_VALUE(3) := '1';
    elsif INIT = '0' then REG_VALUE(0) := '1';
    REG_VALUE(1) := '0';
    REG_VALUE(2) := '0';
    REG_VALUE(3) := '1';
    else -- actual shifting or loading process
      if SHLD = '0' then -- parallel loading
        PARALLEL_LOAD: for I in 0 to NBITS-1 loop
          if PI(I) /= '0' and PI(I) /= '1' then
            INVALID := 1;
          exit PARALLEL_LOAD;
          elsif PI(I) = '1' then REG_VALUE(I) := '1';
          else REG_VALUE(I) := '0';
          end if;
        end loop PARALLEL_LOAD;
      else -- beginning of shifting part
        if RL = '0' then -- shift right
          for I in REG_VALUE'MODEL_RANGE loop
            if I /= 0 then
              REG_VALUE(I-1) := REG_VALUE(I);
            end if;
          end loop;
          REG_VALUE(NBITS-1) := LE;
        else -- left-shift
          for I in REG_VALUE'MODEL_RANGE loop
            if I /= NBITS-1 then
              REG_VALUE(I+1) := REG_VALUE(I);
            end if;
          end loop;
          REG_VALUE(0) := RE;
        end if;
        end if;
        end if;
        end if;
        end if;
      end if;
    end process REGISTER_PROCESS;
  end BEHAVIOR;

List 5. Synthesizable Model Based on the Subset A

-- Note: 'X01Z' is used for the base type for this example
-- VHDL model for 4-bit Parallel-loaded Bidirectional Shift
-- Register with Synchronous Preset, Preclear and INIT
-- controls

Library ...
Use ...

entity REG10 is
generic(NBITS: integer := 4);
  port( CK, PS, PC, INIT, SHLD, RL, LE, RE: in X01Z;
       PI: in X01Z_vector(0 to NBITS-1);
       Q: out X01Z_vector(0 to NBITS-1));
end REG10;

architecture BEHAVIOR of REG10 is
begin
  REGISTER_PROCESS: process
    variable REG_VALUE: X01Z_vector(NBITS-1 downto 0);
    variable INVALID: INTEGER;
  begin
    wait until CK'event and CK='1'; -- rising-edge trigger
    -- check input
    INVALID := 0;
    if INIT /= '0' and INIT /= '1' then INVALID := 1;
    elsif PS /= '0' and PS /= '1' then INVALID := 1;
    elsif PC /= '0' and PC /= '1' then INVALID := 1;
    elsif SHLD /= '0' and SHLD /= '1' then INVALID := 1;
    elsif (RL /= '0' and RL /= '1') then INVALID := 1;
    -- initialization
    elsif PC = '0' then REG_VALUE(0) := '0';
    REG_VALUE(1) := '0';
    REG_VALUE(2) := '0';
    REG_VALUE(3) := '0';
    elsif PS = '0' then REG_VALUE(0) := '1';
    REG_VALUE(1) := '1';
    REG_VALUE(2) := '1';
    REG_VALUE(3) := '1';
    elsif INIT = '0' then REG_VALUE(0) := '1';
    REG_VALUE(1) := '0';
    REG_VALUE(2) := '0';
    REG_VALUE(3) := '1';
    else -- actual shifting or loading process
      if SHLD = '0' then -- parallel loading
        PARALLEL_LOAD: for I in 0 to NBITS-1 loop
          if PI(I) /= '0' and PI(I) /= '1' then
            INVALID := 1;
          exit PARALLEL_LOAD;
          elsif PI(I) = '1' then REG_VALUE(I) := '1';
          else REG_VALUE(I) := '0';
          end if;
        end loop PARALLEL_LOAD;
      else -- beginning of shifting part
        if RL = '0' then -- shift right
          for I in REG_VALUE'MODEL_RANGE loop
            if I /= 0 then
              REG_VALUE(I-1) := REG_VALUE(I);
            end if;
          end loop;
          REG_VALUE(NBITS-1) := LE;
        else -- left-shift
          for I in REG_VALUE'MODEL_RANGE loop
            if I /= NBITS-1 then
              REG_VALUE(I+1) := REG_VALUE(I);
            end if;
          end loop;
          REG_VALUE(0) := RE;
        end if;
        end if;
        end if;
        end if;
        end if;
      end if;
    end process REGISTER_PROCESS;
  end BEHAVIOR;

239
elsif PC /= '0' and PC /= '1' then INVALID := 1;
elsif SHLD /= '0' and SHLD /= '1' then INVALID := 1;
elsif ( RL /= '0' and RL /= '1') then INVALID := 1;
-- initialization
elsif PC = '0' then REG_VALUE(0) := '0';
REG_VALUE(1) := '0'; REG_VALUE(2) := '0';
REG_VALUE(3) := '0';
elsif PS = '0' then REG_VALUE(0) := '1';
REG_VALUE(1) := '1'; REG_VALUE(2) := '1';
REG_VALUE(3) := '1';
elsif INIT = '0' then REG_VALUE(0) := '1';
REG_VALUE(1) := '0'; REG_VALUE(2) := '0';
REG_VALUE(3) := '1';
else -- actual shifting or loading process
if SHLD = '0' then -- parallel loading
PARALLEL_LOAD: for I in 0 to NBITS-1 loop
if PI(I) /= '0' and PI(I) /= '1' then
   INVALID := 1;
   exit PARALLEL_LOAD;
elsif PI(I) = '1' then
   REG_VALUE(I) := '1';
else
   REG_VALUE(I) := '0';
end if;
end loop PARALLEL_LOAD;
else -- beginning of shifting part
if RL = '0' then -- shift right
for I in REG_VALUE\'REVERSE_RANGE loop
if I /= 0 then
   REG_VALUE(I-1) := REG_VALUE(I);
end if;
end loop;
REG_VALUE(NBITS-1) := LE;
else -- left-shift
for I in REG_VALUE\'RANGE loop
if I /= NBITS-1 then
   REG_VALUE(I+1) := REG_VALUE(I);
end if;
end loop;
REG_VALUE(0) := RE;
end if; -- end of RL
end if; -- end of SHLD
end if; -- the end of check input
-- output assignment
-- output Q
if INVALID = 1 then
   Q <= (others => 'X');
else
   for I in 0 to NBITS-1 loop
      if REG_VALUE(I) = '1' then
         Q(I) <= '1';
      else
         Q(I) <= '0';
      end if;
   end loop;
end if;
end process REGISTER_PROCESS;
end BEHAVIOR;

List 6. Synthesizable Model Based on the Subset B

7. Conclusion

This paper provides a new approach to overcome two problems in the automatic generation of synthesizable VHDL models: the synthesizable subset of VHDL and the multiple subsets. This approach is based on the idea of independent and user-definable synthesis rule sets. The model generator described in this paper takes the user-specified engineering parameters and the user-defined synthesis rule set as inputs. It is shown that the model generator successfully generates a synthesizable VHDL model based on the user-defined synthesis rule set as an output. By changing or creating the synthesis rule set, a designer is able to generate a synthesizable VHDL model for different synthesizable subsets.

8. References


