Rule-based and Configurable VHDL Synthesizability Checking for Synthesis-oriented Design

Choon B. Kim
DAZIX, An Intergraph Company
m/s LR23B2
One Madison Industrial Park Huntsville, AL 35894-0001
Tel: (205)730-8704 FAX: (205)730-8344 e-mail: choon@mug.b23b.ingr.com

Abstract

Currently, no synthesizer supports the full set of VHDL. The synthesizable VHDL subset varies depending on the synthesizer. Therefore, a designer needs to know the details of the synthesizable VHDL subset and the synthesis policy set by a particular synthesizer. This paper presents a development of VHDL Synthesizability Checker(VSC) which performs a rule-based, configurable VHDL synthesizability checking. It reads a VHDL model, checks the synthesizability of the model, and generates a report. Unlike a checker within a synthesizer, VSC performs the checking process based on a user-defined rule set. By separating the user-defined rule part and the synthesizability checking part, a designer can configure his/her own synthesis rule set, and can handle the different synthesizable VHDL subset.

1. Introduction

The support for the synthesis-oriented design has grown considerably during the last several years[1-6]. Among various phases of the synthesis-oriented design, the synthesis phase is the most critical step for two reasons. First, many synthesis tools assume that a designer simulates his/her design before going through the synthesis phase. Therefore, a designer tends to use the full set of VHDL for his/her design. However, currently no synthesis tool supports the full set of VHDL. The VHDL model may be rejected during the synthesis phase. This may force the designer to go back to the starting point of design, and to repeat the phases (e.g., model modification, simulation, and resubmission to synthesizer) until his/her design is accepted by the synthesizer. Second, a synthesis tool supports a synthesizable subset of VHDL normally defined by the tool manufacturer. A designer should know the details of this subset of VHDL and the synthesis policy for passing the synthesis phase successfully. When the synthesizable subset is changed by either the revision of the synthesizer or by using different vendor's synthesizer, all changes in the synthesizable subset of VHDL and the different synthesis policies must be considered carefully in the design.

Therefore, it is desirable to incorporate a synthesizablity checking in the early stage of design. Practically, a designer may want to check the synthesizability of the VHDL model even before the simulation in order to avoid the design time wasted for repeating the model_modification-simulation-synthesis cycle. The goal of this research is to develop a solution that can satisfy such a need. VHDL Synthesis Checker(VSC) described in this paper reads a VHDL model, checks the synthesizability of the model, and generates a report. For handling different synthesizable subsets, the rule specification part and the synthesizability checking part are separated.

2. Overall Flow of VHDL Synthesizability Checking Process

VSC reads a VHDL model, checks the synthesizability of the model, and generates a report. The overall flow is depicted in Fig. 1. As it can be seen in the figure, the whole process is divided into two phases: Rule Abstraction and Synthesizability Checking.

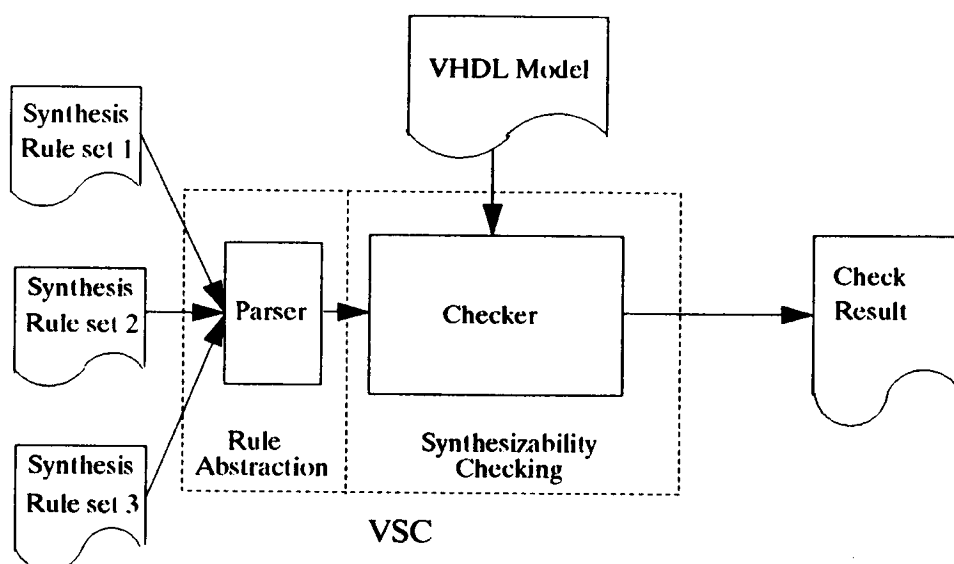

Fig. 1 Overall Flow of Synthesizability Checking Process

The synthesis rules are described in a separate file by using the Rule Specification Format(RSF). In the Rule Abstraction phase, the contents in an external rule file are parsed first. The synthesis rules are abstracted from the parsing process. As a result, a filter is created based on the synthesis rules abstracted. During the Synthesizability Checking phase, every construct in an input VHDL model is checked against the rules abstracted from the previous phase. Any rule violation is detected and reported as the input VHDL model is passed through the filter. VSC pinpoints the location of the violations with the error messages which give the level of violation as well as the nature of the violation.

A user provides a VHDL model and a synthesis rule set. A synthesis rule is encoded by using the Rule Specification Format(RSF) which will be explained in the next section.

3. Rule Specification Format(RSF)

A synthesis rule described in English needs to be encoded for the rule abstraction. Due to the high complexity of the grammar and the semantics associated with a natural language, like English, some simplification is needed for encoding. There may be many different ways of encoding methods. A natural language parsing can be used, too. A relatively simple but efficient method, using a record structure with several fields, was developed for this study.

As shown in Fig. 2, a synthesis rule is represented as a record which has 11 fields: region, statement, statement_set, statement_location, object, object_set, attribute, attribute_set, exception, others, and synthesizability. Each field has a finite numbers of values. Note that the field name(e.g., object) is not related to the definition in VHDL(object)[7], rather, it was defined for RSF purposes only. The region field represents the place information for a rule. The main criteria used is the library_units of VHDL. For example, the valid values for the region field are {none, context_clause, entity_declaration, configuration_declaration, package_declaration, architecture_body, package_body}. The 'none' is a don't care condition which is used when the rule does not specify the place information. The valid values of each field were determined based primarily on the currently available synthesizers. They are summarized in List 1.

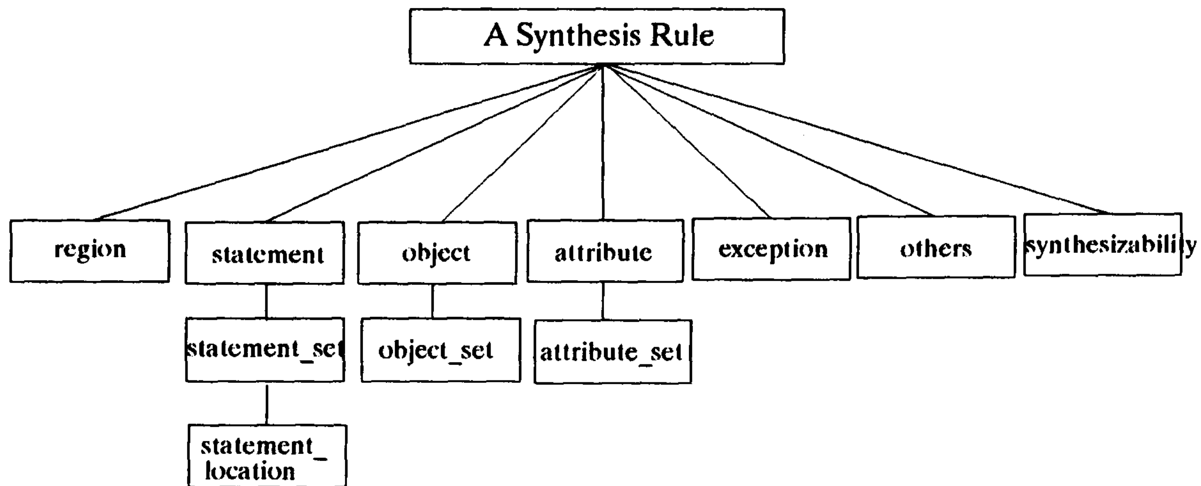

Fig. 2 A Synthesis Rule Encoding

List 1. A list of the values of each field of a record in RSF (The values are listed in alphabetical order.)

the value of region field = {architecture_body, configuration_declaration, context_clause, entity_declaration, none(default), package_body, package_declaration}

the value of statement field = {array, assertion, attribute_specification, block, component_declaration, configuration_specification, declaration, delay_specification, entity_statement_part, generic_clauses, guarded_signal_assignment, if_statement, multiple_waveform, none(default), port_clauses, process, subprogram, type_declaration, use_clauses, use_of, wait_for, wait_on, wait_only, wait_statement, wait_until}

the value of statement_set field = {complement, normal(default)}

the value of statement_location field = {first, not_first, none(default)}

the value of object = {access_type, alias, allocator, constant, file_type, floating_point_type, generic, guard, incomplete_type, multiple_index, none(default), null_array, null_range, null_slice, operator, parameter, physical_type, port, predef_attr_set, record_type, selected_qualified_name, sensitivity_list, signal, signal_index, userdef_attr_set, std, work, use_of_access_type, use_of_file_type, use_of_floating_type_const, use_of_physical_type, use_of_record_type, variable}

where, predef_attr_set ::= {[predef_attr_base] [predef_attr_event] [predef_attr_high] [predef_attr_left] [predef_attr_length] [predef_attr_low] [predef_attr_range] [predef_attr_reverse_range] [predef_attr_right] [predef_attr_stable]},

userdef_attr_set ::= {[userdef_attr_arrival] [userdef_attr_rise_arrival] [userdef_attr_fall_arrival] [userdef_attr_max_rise_arrival] [userdef_attr_max_fall_arrival] [userdef_attr_min_rise_arrival] [userdef_attr_min_fall_arrival] [userdef_attr_drive] [userdef_attr_fall_drive] [userdef_attr_rise_drive] [userdef_at-

tr_equal] [userdef_attr_opposite] [userdef_attr_logic_one] [userdef_attr_logic_zero] [userdef_attr_dont_touch_network] [userdef_attr_load] [userdef_attr_unconnected] [userdef_attr_max_area] [userdef_attr_max_transition] [userdef_attr_max_delay] [userdef_attr_max_fall_delay] [userdef_attr_max_rise_delay] [userdef_attr_min_delay] [userdef_attr_min_fall_delay] [userdef_attr_min_rise_delay]}

the value of object_set field = {complement, normal(default)}

the value of attribute field = {default_value, initial_value, integer, none(default)}

the value of attribute_set field = {complement, normal(default)}

the value of exception field = {none(default), user_defined_attribute}

the value of others field = {none(default), power_of_2, two}

the value of synthesizability field = {ignored, supported(default), translated, unsupported}

The statement field represents a target statement to which a rule is applied. The statement_set field indicates whether a rule is applied to the target statement or to the statements other than the target statement. The statement_location field shows whether the target statement is at a particular location (e.g., first within a process). The object field represents a particular object within a target statement. The object_set field indicates whether a rule is applied to the target object or to objects other than the target object. The attribute field represents a particular attribute or property associated with a target object. The attribute_set field indicates whether a rule is applied to the target attribute or to attributes other than the target attribute. The exception field is used to handle a situation where a particular type is unsupported except when it is used with user-defined attributes. The others field is reserved to handle a situation that is not covered by any other field.

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

Syntax for RSF

Notes for the use of RSF:
- A comment in RSF is the same as in VHDL.
- An identifier in RSF is the same as in VHDL.
- Case sensitive
- Reserved Words: rule, is, a set of field names,
  a set of field-value names, a set of VHDL reserved words

```
Rule_Specification_File ::= synthesis_rule {synthesis_rule}
synthesis_rule ::= rule rule_number is '{' rule_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 | synthesizability
field_value ::= { all the values listed in List 1}
```

As an example, several synthesis rules are encoded as follows.

```
rule 1 is
-- "Initial values are unsupported in a signal declaration."
{
   statement: declaration;
   object: signal;
   attribute: initial_value;
   synthesizability: unsupported;
} 1;

rule 2 is
-- "the use of floating-point type is unsupported except floating-point constants
-- used with user-defined attributes."
{
   object: use_of_floating_type_const;
   exception: user_defined_attribute;
   synthesizability: unsupported;
} 2;

rule 3 is
-- "the use of floating-point type is unsupported except floating-point constants
-- used with user-defined attributes."
{
   object: use_of_floating_type_const;
   exception: user_defined_attribute;
   synthesizability: unsupported;
} 3;

rule 4 is
-- "the wait statement, if used, must be the first statement in a process"
{
   statement: wait_statement;
   statement_location: not_first;
   synthesizability: unsupported;
} 4;
```

4. Checking Process

The checking process starts with parsing the user-defined ruleset by a (rule) Parser shown in Fig.1. As a result, a rule data structure which holds the encoded rules is created in the memory. The data structure has an interface function, rule_match(argument_list). The rule_match function, when invoked with proper arguments, searches the rule data structure to find a rule record which matches the input arguments. If successful, the function returns a rule number and the synthesizability information. The argument_list of the function is the value of the eleven fields described in section 3, i.e.,

argument_list = arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9, arg10

[input] arg0 = the value of region field
[input] arg1 = the value of statement field
[input] arg2 = the value of statement_set field
[input] arg3 = the value of statement_location field
[input] arg4 = the value of object field

[input] arg5 = the value of object_set field
[input] arg6 = the value of attribute field
[input] arg7 = the value of attribute_set field
[input] arg8 = the value of exception field
[input] arg9 = the value of others field
[output] arg10 = the value of synthesizability field

Note that arg10 is an output parameter from the function, and all other arguments are input parameter to the function.

After a rule data structure is created, the Checker shown in Fig.1 starts parsing a VHDL model source code. During the parsing, a context-sensitive environment table is maintained for correct function argument setting. For example, with help from this table, a condition such as the first statement within a process or the qualified name outside of the use clause can be recognized. The rule_match function is invoked whenever either a VHDL statement is recognized or a special keyword is lexically scanned. The algorithm CHECKING describes the whole checking process.

algorithm CHECKING:

```
1. start of VSC
2. if (a rule file is not specified/readable)    exit with error message
3. while read(rule of a rule file) = not EOF
4.  { create a rule data structure }
5. if (a VHDL source file is not specified/readable)   exit with error message
6. while read(token of a VHDL source file) = not EOF
7.  {
8.    update context-sensitive environment table;
9.    determine the arguments;
10.  if (token is a special keyword or a VHDL statement is recognized)
11.     {
12.        invoke the rule_match function;
13.      if (matching rule found)
14.          report the synthesizability checking result;
15.     }
16.  }
17. report the summary of checking
18. exit
```

5. Example

A VHDL model is presented for the synthesizability checking example. The model has been intentionally corrupted to show the checking capability of VSC. Some other unnecessary constructs were added for the same reason. The check result is followed after the model. The check result consists of a level code (E or W), a line number, an explanation of the violation, and a corresponding rule number. The level code E indicates a fatal error (e.g., an unsupported construct) while the level code W indicates a warning (e.g., an ignored construct). The rule number points to the rule which was used as a basis for checking. User can control the checking process by changing the contents of the rule. Needless to say, the model source and the check result can be tightly integrated when a VHDL syntax-driven editor is used.

[VHDL Model Source -- Moore-type FSM model]

```
-- The model has been intentionally corrupted for synthesizability checking example.

Use work.all;
Use mypackage.all;

entity FSM is
  generic(RESET_DELAY: time := 10 ns);
  port ( INPUT_TO_FSM, CK, RESET   : in  Bit;
         OUTPUT_FROM_FSM : out Bit );
 begin
  assert RESET = '0' report "RESET error";
end FSM;

architecture BEH of FSM is
  type STATE_TYPE is (STATE1, STATE2);
  signal  CURRENT_STATE,  NEXT_STATE, TEMP_STATE: STATE_TYPE;
  signal  BUS_SIGNAL: RBit register;
  signal S1: bit := '1';
  alias ANOTHER_CK: Bit is CK;
  attribute STATE_NO: integer;
  attribute STATE_NO of CURRENT_STATE: signal is 1;
 begin
  S1 <= transport ANOTHER_CK;
  P1: process begin
    wait on CK for 100 ns;
    CURRENT_STATE <= TEMP_STATE;
    wait until CK'event and CK = '1';
    CURRENT_STATE <= NEXT_STATE;

    while (CURRENT_STATE /= STATE1 and CURRENT_STATE /= STATE2) loop
       CURRENT_STATE <= NEXT_STATE;
     end loop;
  end process P1;

  P2: process(CURRENT_STATE, INPUT_TO_FSM)
      constant NO_CHANGE: integer := 0;
      variable VAR1 : integer := 0;
  begin
     case  CURRENT_STATE  is
        when STATE1 =>
            OUTPUT_FROM_FSM <= '0' after 10 ns;
             if (INPUT_TO_FSM = '0')  then  NEXT_STATE <= STATE1;
             else  NEXT_STATE <= STATE2;
             end if;
        when STATE2 =>
            OUTPUT_FROM_FSM <= '1' after 10 ns;
             if (INPUT_TO_FSM = '0')  then  NEXT_STATE <= STATE2;
             else  NEXT_STATE <= STATE1;
             end if;
     end case;
  end process P2;
 end BEH;
```

[Synthesizability Checking Result]
Note: E = Error, W = Warning, Ln = Line number, (Rule n) = Synthesis Rule number

E L4: separate compilation for "mypackage" is unsupported(Rule 5)
E L7: Generics in entity declarations are unsupported(Rule 2)
W L10: The entity statement part is ignored(Rule 1)
E L17: "register" declaration is unsupported(Rule 21)
E L18: initial value(signal) is unsupported(Rule 22)
W L19: Alias is ignored(Rule 28)
W L20: user-defined attribute can be specified, but user-defined attributes at any place else in the VHDL source are unsupported(Rule 28)
W L21: User-defined attribute can be specified, but user-defined attributes at any place else in the VHDL source are unsupported(Rule 28)
W L23: "transport" is ignored (Rule 44)
E L26: This wait statement is unsupported(Rule 402)
E L28: The wait statement, if used, must be the first statement in a process(Rule 41)
E L31: While loop is unsupported(Rule 46)
W L36: sensitivity lists in process statements are ignored(Rule 48)
E L38: initial value(variable) is unsupported(Rule 25)
W L42: the use of physical type is ignored in delay specification(Rule 8)
W L48: the use of physical type is ignored in delay specification(Rule 8)

6. Conclusion

When a VHDL synthesis is used in his/her design, a designer may face many difficulties. One of them is the fact that, currently, no synthesizer supports the full set of VHDL. The synthesizable VHDL subset varies depending on the synthesizer. Therefore, a designer needs to know the details of the synthesizable VHDL subset and the synthesis policy set by a particular synthesizer. This paper presents an approach to solve that problem. The VHDL Synthesizability Checker(VSC), described in this paper, reads a VHDL model, checks the synthesizability of the model, and generates a report. Unlike a checker within a synthesizer, VSC performs the checking process based on a user-defined rule set. By separating the user-defined rule part and the synthesizability checking part, a designer can configure his/her own synthesis rule set, and can handle the different synthesizable VHDL subset. A sample VHDL source and the result of checking is provided to show the capability of the tool.

7. References

[1] "Experience in VHDL Model Verification: Pre-Synthesis and Post-Synthesis" VIUF Fall, 1991, pp87 Carl Cleaver, Mike Derr
[2] "VHDL Synthesized CSP Systems" VIUF Fall, 1991, pp95 , Richard J. Auletta
[3] "Design Automation Takes Over More Tasks Early On", Electronic Design, June 13, 1991 pp47 Lisa Maliniak
[4] "Are designers' expectations for synthesis realistic?", Computer Design, Nov. 1, 1990 pp 110, Barbara Tuck
[5] "Logic synthesis fine tunes abstract design descriptions", Computer Design, June 1, 1990, pp84, Ernest Meyer
[6] "High Level Design: shortcut or detour?", ASIC Technology & News, April, 1991 Bill Groves
[7] VHDL Language Reference Manual, IEEE Std 1076-1987