Towards Object-Oriented Extensions to VHDL for Effective Reuse of Models and Components

Norihiko Yoshida
Department of Computer and Information Sciences
Nagasaki University
Nagasaki 852-8521, Japan
yoshida@cis.nagasaki-u.ac.jp

Toshiomi Moriki
Central Research Laboratory
Hitachi, Ltd.
Kokubunji, Tokyo 185-8601, Japan
t-moriki@crl.hitachi.co.jp

Abstract

Some OO extensions to HDL's, such as VHDL and SFL, have been proposed. However, theoretical researches have revealed a problem of inheritance anomaly (IA) which arises when inheritance is applied to concurrent systems. Hardware systems are concurrent systems, therefore the IA problem is inevitable also in the OO design of hardware systems. In this paper, we investigate a certain class of the IA problem in OO extensions to VHDL, and proposes a framework to avoid it.

1. Introduction

The object-oriented (OO) paradigm is now indispensable for software design and development. Likewise, it is no doubt that the paradigm will be indispensable also for hardware design, hardware/software co-design and system level design in the near future. Consequently, some OO extensions to HDL's, such as VHDL and SFL, have been proposed [1-9].

Inheritance is one of the fundamental features of the OO paradigm. It enables programmers to incrementally add properties to objects, and therefore facilitates reuse of models and components. However, theoretical researches have revealed a difficult problem which arises when inheritance is applied to concurrent systems. Additional properties of a subclass may cause undesirable re-definitions of the existing properties in the superclass. Instead on being able to incrementally add properties, the programmer may be required to re-define some (or, in the worst case, all) inherited properties, therefore the benefits of inheritance are lost. This problem is called *inheritance anomaly* (IA) [10,11].

Hardware systems are concurrent systems, therefore the IA problem is inevitable also in the OO design of hardware systems. Among researches on OO extensions to HDL's, Schumacher et al. [7] and we [8,9] studied this problem. In this paper, we investigate a framework to avoid the IA problem in OO extensions to VHDL.

2. OO Extensions to HDL's

Among HDL's, VHDL is the most vigorously researched towards OO extensions [1-7]. OO-VHDL Study Group has already been organized under IEEE DASC since 1993. As VHDL is a "big" language, proposals for OO extensions to VHDL have been categorized into:

- Type-based (or record-based) extensions,
- Entity-based extensions,
- Abstract class extension.

Among them, Ashenden is proposing SUAVE, a type-based (Ada95-like) OO extension to VHDL, incorporating some communication abstraction [4]. Nebel et al. is proposing a class-based OO extension, Objective VHDL [5,6,7]. These two are now considered as prospective candidates for future IEEE standard.

There has been no attempt, as far as we know, towards OO extensions to Verilog-HDL. We studied an OO extension to yet another and very simple HDL, SFL [8].

Below are very simple example programs in SUAVE and Objective VHDL. Both define an object class for a `counter`. The SUAVE example defines an active object in the sense that the object decides by itself how to respond to inputs, while the Objective VHDL example defines a passive object in the sense that the object simply responds to procedure calls.

[SUAVE]

```
entity counter is
  generic ( type count_type is ( <> ) );
  port ( clk : in bit; data : out
count_type );
end entity counter;
architecture behavioral of counter is
begin
```

```
  count_behavior : process is
    variable  count :
      count_type := count_type'low;
  begin
    data <= count;
    wait until clk = '1';
    if count = count_type'high then
      count := count_type'low;
    else
      count := count_type'succ( count );
    end if;
  end process count_behavior;
end architecture behavioral;
```

[Objective VHDL]

```
type counter is class
  class attribute value: integer
  function status return integer;
  for signal, variable
    procedure count_up;
    procedure reset;
    procedure load (i: integer);
  end for;
end class counter;

type counter is class body
  function status return integer is
  begin
    return value;
  end;
  for signal, variable
    procedure count_up is
    begin
      load (value + 1);
    end;
    procedure reset is
    begin
      load (0);
    end;
  end for;
  for signal
    procedure load (i: integer) is
    begin
      value <= i;
    end;
  end for;
  for variable
    procedure load (i: integer) is
    begin
      value := i;
    end;
  end for;
end class body counter;
```

3. Inheritance Anomaly

Now we summarize researches on the problem of inheritance anomalies (IA's) briefly. Matsuoka et al. investigated the problem in single-thread concurrent objects [10,11], and Thomas investigated in multi-thread concurrent objects [12]. Matsuoka et al. introduced some typical classes of IA's such as:

- State Partitioning,
- History-Only Sensitiveness,
- State Modification;

and also introduced a set of standard benchmark examples using programs of bounded buffers. Almost all the researches thereafter aimed to avoid the IA's in these benchmarks. However, the problem is still only vaguely defined. There has been no formal proof yet that the classes are exhaustive (there could be other undiscovered types of IA's). Some formal treatment was attempted recently [13].

The cause of the IA problem is, in short, that the unit of inheritance, i.e. a method, contains codes for synchronizations and codes for behaviors together. Therefore, the most promising solution to avoid the problem is to divide the unit of inheritance into finer something than the method. Simply dividing a method into finer methods cannot be the solution in most cases, because a method is also the atomic execution unit (a method runs in a mutually exclusive manner), and dividing the method breaks the atomicity. Instead of method-based inheritance, there have been some proposals of new OO language constructs for inheriting codes for synchronizations and codes for behaviors separately. From more general standpoint of views, some frameworks for separation of different aspects of codes have also been proposed such as aspect-oriented programming [14].

It must be mentioned, however, that none of widely-used OO languages with concurrency, Java for example, provides yet any mechanism to avoid IA's. The new language constructs for avoiding IA's are often complicated, and may degrade programmer-friendliness of the languages. The two essential concern in designing programming languages in general are expressiveness and programmer-friendliness, which are sometimes conflicting mutually.

4. Inheritance Anomaly in OO-VHDL

Hardware systems are concurrent systems, therefore the inheritance anomaly problem is inevitable in the OO design of hardware systems. Schumacher et al. studied synchronization-related IA in their OO extension [7], as well as analyzing IA in Ada95 [15], and claimed that their OO extension, as is, solved synchronization-related IA's. Their investigation for avoiding IA's followed Ferenczi's framework, which uses nested conditional critical regions [16].

Now we investigate a possibility of another kind of IA in OO extensions to VHDL, especially found in active

object definitions. It relates to active dispatching within `process` definitions. Below is a very simple (in fact, trivial) example of a `resettable_counter`, following the `counter` example shown above. Here we use the SUAVE framework, however the situation is the same also in the Objective VHDL framework.

```
entity resettable_counter is
  generic ( type count_type is ( <> ) );
  port ( clk : in bit;  reset : in bit;
         data : out count_type );
end entity resettable_counter;
architecture behavioral of
resettable_counter is
begin
  resettable_count_behavior : process is
    variable count
      count_type := count_type'low;
  begin
    data <= count;
    wait until reset = '1' or clk = '1';
    if reset = '1' then
      count := count_type'low;
    elsif count = count_type'high then
      count := count_type'low;
    else
      count := count_type'succ( count );
    end if;
  end process resettable_count_behavior;
end architecture behavioral;
```

The program codes presented in the plain font is exactly the same as the definition of a `counter`, while the codes presented in the bold font is to add functionality for `reset`. Therefore, we would be happy if we could define the `resettable_counter` as a derived object class of the `counter` using inheritance, just adding the functionality for `reset`, as something like:

```
entity resettable_counter
  extends counter with
  port ( reset : in bit );
end entity resettable_counter;
architecture behavioral of
resettable_counter extends counter with
begin
  count_behavior : process is
  begin
    wait until reset = '1';
    if reset = '1' then
      count := count_type'low;
    end if;
  end process count_behavior;
end architecture behavioral;
```

We can do it for the `entity` definition; however, we cannot do it for the `architecture` definition actually. We must redefine the whole `process` definition. It is because the `process` statement is indivisible. This is considered as a kind of IA.

5. Proposal for New Language Constructs

Now we introduce new language constructs for avoiding such kind of IA's. The principle behind is to reorganize the `wait` statement and the `if` statement into the guarded-command style of "wait for a condition to stand, and then do something relevant".

The language constructs are combination of: (1) an extended `wait` statement, (2) an `alternation` sub-program, and (3) `before`/`after` prefixes. Before describing each, we present an example using the new constructs (in the bold font) to show an outline of our proposal.

```
architecture behavioral of counter is
begin
  count_behavior : process is
    variable count :
      count_type := count_type'low;
  begin
    data <= count;
    count_behavior_switch( count );
  end process count_behavior;
  alternation count_behavior_switch
    (count : inout count_type) is
  begin
    wait until clk = '1' do
      if count = count_type'high then
        count := count_type'low;
      else
        count := count_type'succ( count );
      end if;
    end wait;
  end alternation count_behavior_switch;
end architecture behavioral;
architecture behavioral of
resettable_counter extends counter with
begin
  alternation count_behavior_switch
    (count : inout count_type) is
  before begin
    wait until reset = '1' do
      count := count_type'low;
    end wait;
  end alternation count_behavior_switch;
end architecture behavioral;
```

Here, the architecture of the `resettable_counter` only defines the functionality for `reset`, as we desire.
(1) Extended `wait` statement: A compound statement

"`wait` <condition> `do` <statement> `end;`" is semantically equivalent to "`wait` <condition> `; if` <condition> `then` <statement> `end;`". The statement is executable (and the entire compound statement is said to be executable) when the condition is fulfilled.

(2) `Alternation` sub-program: An `alternation` sub-program is defined just like a `procedure` or a `function`. An executable statement in the sub-program is chosen arbitrarily and executed. If there is no executable statement at the moment, a system waits until any statement is executable. The order of statements within an `alternation` sub-program is meaningless.

(3) `Before`/`after` prefixes: These are used with inheritance of sub-programs. If no prefix is given in an inherited class (super class or base class) nor an inheriting class (sub-class or derived class), the sub-class definition replaces the super class definition. If a prefix is given, the sub-class definition is added before or after the super class definition. A `before` prefix and an `after` prefix, when applied to an `alternation` sub-program, have the same effect, because the order of statements is meaningless.

The extended `wait` statement and the `alternation` sub-program together construct a guarded command structure, which are found in many concurrent programming languages including Ada. The `before`/`after` prefixes customize inheritance; they were first introduced in the language "Flavors" [17], and found in some multiple-inheritance-based OO programming languages. All these working together enable us to inherit, override or add guarded commands for active dispatching.

The above definition of `resettable_counter` is a sub-class of `counter`. When compiled, it is extended as:

```
architecture behavioral of
resettable_counter is
begin
  count_behavior : process is
    variable count
      : count_type := count_type'low;
  begin
    data <= count;
    count_behavior_switch( count );
  end process count_behavior;
  alternation count_behavior_switch
    (count : inout count_type) is
  begin
    wait until reset = '1' do
      count := count_type'low;
    end wait;
    wait until clk = '1' do
      if count = count_type'high then
        count := count_type'low;
      else
        count := count_type'succ( count );
      end if;
    end wait;
  end alternation count_behavior_switch;
end architecture behavioral;
```

The body of `count_behavior_switch` is translated into a conventional VHDL definition and unfolded within the `process` definition as:

```
count_behavior : process is
  variable count :
    count_type := count_type'low;
begin
  data <= count;
  wait until reset = '1' or clk = '1';
  if reset = '1' then
    count := count_type'low;
  elsif clk = '1' then
    if count = count_type'high then
      count := count_type'low;
    else
      count := count_type'succ( count );
    end if;
  end if;
end process count_behavior;
```

This is semantically equivalent to the definition of the `resettable_counter` shown in the "Inheritance Anomaly in OO-VHDL" section.

6. Concluding Remarks

We proposed new language constructs for avoiding a certain kind of inheritance anomalies in object-oriented extensions to VHDL. Objects in OO-VHDL is either active or passive. The former includes a definition for active dispatch mechanism within itself, and decides by itself how to respond against the input. The latter is composed of some procedure (method) definitions, and simply responds to procedure calls. Our proposal is concerned with inheritance of dispatch mechanisms, typically constructed by `wait`, `process` and `if` statements in VHDL. Such active dispatch mechanisms are widely found in many VHDL programs, and our proposal is effective for OO extensions to such programs.

VHDL is already a complicated language. We could introduce more effective, yet more complex language constructs, or change the VHDL syntax and semantics drastically; however, so as not to make VHDL even more complicated, we considered simplicity as the first importance. In our humble opinion, given any solution proposal for IA's, no matter how sophisticated, we could make a counter-example (sometimes artificial and trivial) against it, because there has been no formal definition of IA's yet.

We have been aiming not at a perfect solution but at a practical, or in other words, simple and modest solution.

We do not intend at all to claim that SUAVE and Objective VHDL are incomplete or imperfect. We do not claim that our proposal solves all kinds of IA either. Our proposal solves some kind of IA, and can work additionally with the OO extension frameworks of SUAVE or Objective VHDL. Also, our proposal should not be able to apply only to VHDL. It should ideally be able to apply to OO extensions to any HDL's and SLDL's. We are now investigating a framework of OO extensions to Verilog-HDL.

Acknowledgments

This research is partly supported by STARC (Program No. 997). We thank Drs. T. Kozawa, O. Matsumura, N. Hirano and K. Wakabayashi for valuable information and fruitful discussions.

References

[1] S. Swamy, A. Molin, and B. Covnot, "OO-VHDL: Object-Oriented Extensions to VHDL", IEEE Computer, 28:10, 18-26 (1995)

[2] J. Benzakki and B. Djafri, "Object Oriented Extensions to VHDL – The LaMI proposal", Proc, CHDL'97 (1997)

[3] P. J. Ashenden and P. A. Wilsey, "Principles for Language Extension to VHDL to Support High-Level Modeling", TR-03/97, Dept. of Computer Science, Univ. of Adelaide (1997); also as TR-204/05/97/ECECS, Univ. of Cincinnati (1997)

[4] P. J. Ashenden, P. A. Wilsey and D. E. Martin, "SUAVE: Extending VHDL to Improve Data Modeling Support", IEEE Design & Test of Computers, 15:2, 34-44 (1998)

[5] M. Radetzki, W. Putzke-Röming and W. Nebel, "A Unified Approach to Object-Oriented VHDL", J. of Information Science and Engineering, 14, 523-545 (1998)

[6] M. Radetzki, W. Putzke-Röming and W. Nebel, "Objective VHDL: Tools and Applications", Proc. FDL'98, 191-200 (1998)

[7] G. Schumacher and W. Nebel, "Object-Oriented Modelling of Parallel Hardware Systems", Proc. DATE '98 (Design, Automation and Test in Europe), 234-241 (1998)

[8] T. Inuo and N. Yoshida, "SFL++ : Object-Oriented Extension for SFL" (in Japanese), Proc. 9th Parthenon Workshop, 13-20 (1996)

[9] T. Moriki and N. Yoshida, "Inheritance Anomaly in Object-Oriented Extensions to Hardware Description Language VHDL" (in Japanese), Record of 1998 Joint Conf. of Electrical and Electronics Engineers in Kyushu (1998)

[10] S. Matsuoka and A. Yonezawa, "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Language", Research Directions in Concurrent Object-Oriented Programming (G. Agha, P. Wegner and A. Yonezawa, eds.), 107-150 (1993)

[11] S. Matsuoka, K. Taura and A. Yonezawa, "Highly Efficient and Encapsulated Re-Use of Synchronization Code in Concurrent Object-Oriented Languages", Proc. ACM OOPSLA '93, 109-126 (1993)

[12] L. Thomas, "Inheritance Anomaly in True Concurrent Object Oriented Languages: A Proposal", Proc. IEEE TENCON '94, 541-545 (1994)

[13] L. Crnogorac, A. S. Rao and K. Ramamohanarao, "Inheritance Anomaly - A Formal Treatment", Proc. 2nd Int'l Conf. on the Formal Methods in Open Object Distributed Systems, 319-334 (1997)

[14] http://www.parc.xerox.com/spl/projects/aop/

[15] G. Schumacher and W. Nebel, "How to Avoid the Inheritance Anomaly in Ada", Proc. Ada-Europe '98 (1998)

[16] S. Ferenczi, "Guarded Methods vs. Inheritance Anomaly - Inheritance Anomaly Solved by Nested Guarded Method Calls", ACM SIGPLAN Notices, 30:2 (1995)

[17] D. A. Moon, "Object-Oriented Programming in Flavors", Proc. OOPSLA '86, 1-8 (1986)