Object-Oriented Modelling of Parallel Hardware Systems

Guido Schumacher, Wolfgang Nebel
FB 10 – Department of Computer Science
Carl von Ossietzky University Oldenburg
D-26111 Oldenburg, Germany
guido.schumacher@informatik.uni-oldenburg.de

Abstract

Object-oriented techniques like inheritance promise great benefits for the specification and design of parallel hardware systems. The difficulties which arise from the use of inheritance in parallel hardware systems are analysed in this article. Similar difficulties are well known in concurrent object-oriented programming as inheritance anomaly but are not yet investigated in object-oriented hardware design. A solution how to successfully deal with the anomaly is presented for a type based object-oriented extension to VHDL. Its basic idea is to separate the synchronisation code (protocol specification) and the actual behaviour of a method. Method guards which allow a method to execute if a guard expression evaluates to true are proposed to model synchronisation constraints. It is shown how to implement a suitable re-schedule mechanism for methods as part of the synchronisation code to handle the case that a guard expression is evaluated to false.

1. Introduction

Re-use of intellectual property in form of hardware models is an important issue in hardware design. It is necessary to provide appropriate modelling techniques which allow re-use of models on all levels of complexity. Beside the re-use of unchanged models the re-use of incrementally modified models is very interesting.

Several proposals have been made to use inheritance in hardware modelling as an object-oriented technique for incremental modification. Most of these proposals are based on language extensions to VHDL which provide inheritance as a key element [3][4][5][8][12][13][14]. All these language extensions allow the modelling of parallel hardware systems in an object-oriented way. When modelling such concurrent systems it can be discovered that synchronization mechanisms and inheritance often conflict. This phenomenon is known as *inheritance anomaly* in the software world [10]. In worst case it cancels the benefits of inheritance out. There is no solution to the inheritance anomaly in the software domain which provides an efficient implementation.

The phenomenon appears in a similar way in object-oriented hardware modelling. This is analysed in Chapter 4. Chapter 5 introduces a hardware modelling technique which solves the anomaly. It is based on an object-oriented language extension to VHDL[13]. For those readers not familiar with the language extension a short survey is given in Chapter 2. Chapter 3 presents some basic modelling techniques using the language extension. It is explained how to apply inheritance and overloading to methods of concurrent objects.

2. Object-oriented language extension to VHDL

VHDL as it is today allows to a certain extend the modelling of systems in an object-based view [11]. Objects in the definition of the LRM [9] are signals, variables and ports. They can be used to model objects like state variables or interconnections. As these are typed objects it is possible to aggregate complex objects from simple ones by using record types and establish a part-of-hierarchy. In this view a record definition can be used to define the instance variables or attributes of a class [15]. The behaviour of a class (its methods) can be implemented by a subprogram which reads or modifies attributes, i.e. the record elements. Packages are used as an encapsulation mechanism. The class interface is then described by subprogram declarations.

What is missing in this concept to become an object-oriented concept is the possibility to modify a class by an incremental specialization to build up an is-a-hierarchy. This concept of incremental specialization which allows a class to inherit characteristics from a parent class is one of the most fundamental object-oriented features to enable re-use. A language extension to VHDL which adds the missing inheritance concept to types has been presented in [13]. The following very brief summary of the language

extension is limited to those features which are necessary to follow the rest of the paper.

The extension allows to build up an is-a-hierarchy of records. The records used to create the hierarchy are called *tagged records*. It is possible to derive a new tagged record from an existing one. In such a case the new record inherits the structure and the behaviour of the parent record. The inherited structure can be modified by adding new attributes to the class. The parent record of a derived tagged record definition is marked by the keyword new. Any additional new attributes are listed following the keyword with. The derived tagged record inherits all the subprograms which implement the behaviour of the parent record. The interface specification, i.e. the subprogram declaration, is automatically adapted to the derived tagged record. The behaviour of the derived class can be modified by adding new methods to the class.

In addition to this inheritance hierarchy a subtype hierarchy can be created by means of the 'Class attribute which provides the basis for heterogeneous object containers and hence polymorphism.

Invoking a method of a heterogeneous object container means to start a method of that tagged subtype which substitutes the class-wide type at that time of the simulation when the method is invoked. As methods are implemented in the language extension by subprograms which can be inherited and overloaded subprograms, overloading is extended to polymorphism. Please note that different to Ada'95 which knows a similar mechanism no access types are needed to model a heterogeneous object container.

An example is given to illustrate the modelling capabilities provided by the language extension. The example consists of a bounded buffer with a put and a get operation. It can be modelled as a tagged record (see Listing 1).

```
type bbuffer is tagged record
   buf: buffer_array (buffersize);
   buf_in, buf_out: buffersize;
end record;
```

Listing 1 Bounded buffer

The methods are implemented as subprograms (see Listing 2).

```
procedure execute_put(signal this : inout bbuffer;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channel : out
      bufferreturnvalue'class);

procedure execute_get(signal this : inout bbuffer;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channel : out
      bufferreturnvalue'class);
```

Listing 2 Methods

A bounded buffer object is instantiated as a signal of the tagged type "bbuffer". The buffer communicates with objects running in parallel by signals. Messages to the buffer are encoded by tagged records and can be sent to a "bbuffer object" via a signal of a class-wide type "op_channel'class". These messages can be derived from the tagged record "op_channel", e.g. see Listing 3.

```
put is new op_channel with
   operand : bufferelement;
end record;
```

Listing 3 Derived type

A complementary signal of a class-wide type "bufferreturnvalue'class" is used to send results back to the caller of a method. The buffer implementation is sketched in Figure 1. The methods are invoked by sending a message

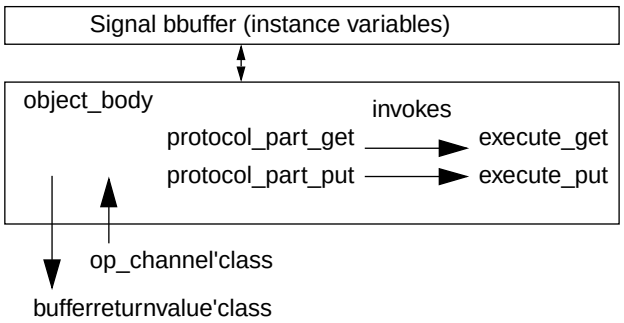


Fig. 1 Bounded buffer implementation

of type "op_channel'class". It can be decoded by a subprogram implementing the protocol code of a method. The corresponding methods are then invoked and executed in a sequential order.

The language features and the implementation style presented in this chapter allow signature compatible modifications of a class as it is known from the software domain. It means that the behaviour of a class which can be incrementally modified is approximated by a signature for modification purposes [15]. The next chapter explains why this approximation often is not precise enough in hardware design. It introduces restrictions which can be imposed on the modelling style to nevertheless successfully use signature compatible modifications.

3. Approximation of behaviour

To substitute a class-wide object by an object of a derived tagged subtype in any context of the class-wide type during simulation requires the compatibility between the class-wide type and its subtype with respect to the structure and behaviour, i.e. the methods. In the language extension the common structure of a parent class and a derived class is preserved and the classes behaviour is inherited. Thus the required subtype relation can be established.

However, the situation is different if the behaviour is modified by overloading an inherited method. In this case the new method has to be compatible with the old one. As

it is normally too difficult to analyse precisely and completely if the behaviour is compatible, only the signatures of the method are checked for compatibility. This approximation of behaviour works very well for many languages and applications in the software domain. At the start of a method a set of values is passed as parameters and after completing the calculations return values are given back to the caller. The signature describes the type and mode (in, inout, out) of the parameters and thus implicitly defines the time when the values have to be passed as parameters. In other words, the signature contains all the relevant protocol information. This is illustrated by an example from the software world. Consider a buffer “x_buf2” which is derived from “bbuffer”. “x_buf2” has the same structure as “bbuffer” but an additional method “get2” which returns the two oldest items from the buffer back to the caller (see Listing 4).

```
procedure execute_get2( this: inout x_buf2;
   op_channel : in bufferoperation'class;
   oldest_item: out bufferreturnvalue'class;
   second_oldest_item: out bufferreturnvalue'class);
```

Listing 4 Signature of get2

For this example it is possible to overload the method by looking at its signature[1] without analysing the body of the method in detail.

In VHDL however, it would be possible to model the “x_buf2” example with a method “execute_get2” which has the same interface list as the method “execute_get” because it would be possible to multiplex the two return values on one return signal. Hence to overload the method it is not sufficient to just know the signatures and to know that the method returns the two oldest items. The protocol has to be analysed how the return values are given back. The protocol has to be preserved if the method is re-defined in a derived class. Unfortunately, there is no compiler technique to check if the protocol is preserved.

The fact that a signature is not sufficient to completely describe a procedure interface is not a problem of object-oriented modification techniques but a more general one of encapsulation and re-use in VHDL. A signature does not describe the fact that an interpretation of a signal’s value may change over time and that not a single value but a projected output waveform is given back to a caller.

A solution to this problem is to impose restrictions on the use of methods, i.e. subprograms. This can be seen as a protocol implicitly defined for each method.

One of the most restrictive protocols is to block a caller of a method until it receives all results. The caller does not change its state before it has all results from the method call. Moreover the caller must not send any parameter values to the method during the method’s execution while the caller is blocked. A projected waveform which schedules a transaction during the method’s execution must not be passed as parameter. Thus the protocol is similar to the one used in most sequential object-oriented languages.

A more powerful but also more complex approach is to allow parallelism without blocking and to try to separate protocol specification and implementation from the functionality of a method. The functionality is modelled in methods which are called inside the methods implementing the protocol. Re-definition of the behaviour often can be done by incrementally modifying the functionality without affecting the existing protocol parts. In such a case a method implementing a protocol can be inherited preserving the compatibility. Again the restriction has to be imposed that projected waveforms passed as parameters of mode in must not schedule transactions during the execution of the method and that projected waveforms passed as parameters of mode out must not schedule transactions after the execution of the method. In cases in which the protocol has to be modified the method’s signature together with its protocol method(s) can be seen as interface which has to be changed. It is not necessary to understand the details of the implementation of the functional part.

In the example the method “protocol_part_get2” could be implemented by a procedure which calls two times the method “execute_get” without allowing the execution of “protocol_part_get2” to be interrupted by another invocation of “get2” or “get”. After the execution of the procedures “execute_get” the result is given back to the caller. Modifying the behaviour without changing the interface consisting of the signature plus the method “protocol_part_get2” is done by just re-defining the methods “execute_get”.

Unfortunately, the separation of protocol and functionality is not so straightforward like in the example in many cases. Often it is necessary to partition the functionality and the protocol into several different methods which makes the solution much more complex.

In the example this would be the case if the protocol allows the execution of one or more put operations in between the two calls of the method “execute_get” in the method “protocol_part_get2”. A possible implementation is sketched in figure 2.

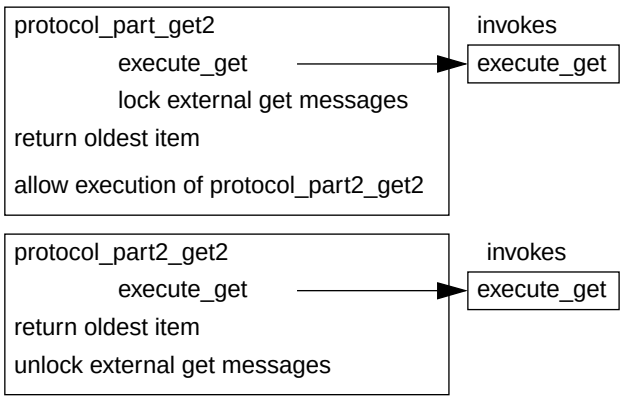


Fig. 2 Implementation 2 of method get2 of x_buf2

1 The term signature is not used in the strict sense of the LRM. Here it also covers the names of the parameter and its modes.

The protocol has to be divided into two parts because the get2 operation can be interrupted by put operations and the result is given back in a sort of multiplexed protocol.

This chapter has shown that a signature is not sufficient to approximate the behaviour of a method for re-use purposes and that the protocol implementation has to be considered as a part of the method's interface. Re-use then can be achieved without breaking the encapsulation by analysing the method's interface and thus reducing the difficulties when inheriting and modifying a protocol.

The next chapter presents some more situations when protocol implementation conflicts with inheritance.

4. Inheritance anomaly

Communication between objects in hardware systems can be defined by all kinds of protocols. A protocol describes among other things which messages can be accepted by an object depending on its own and the communication channel's state. The restriction of acceptable messages is called synchronization constraint.

In the example of the bounded buffer synchronization constraints are to accept a get message only if the buffer is not empty and to accept a put message only if the buffer is not full. So three states of the buffer can be identified which are therefore relevant for specifying and implementing the protocol: "buffer_empty", "buffer_full", and "buffer_partial". The synchronization code which implements the protocol must take the constraints into account.

If a new class is derived from an existing one it might happen that the number of states which have to be distinguished for specifying the synchronization constraints increases.

In the example of the derived class "x_buf2" the state "buffer_partial" has to be partitioned into two new states: "buffer_partial_more_than_one" and "buffer_partial_one". The state "buffer_partial_more_than_one" means that more than one item is stored in the buffer and "buffer_partial_one" means that exactly one element is stored in the buffer. Now, the message get is accepted when the buffer is in state "buffer_partial_more_than_one" or in state "buffer_partial_one". The message "get2" is accepted when the buffer is in state "buffer_partial_more_than_one".

As a consequence of the state partitioning the protocol code of the derived class which contains the synchronization constraints has to be modified without becoming incompatible with the protocol of the parent class. This may cause problems depending on the implementation technique of the protocol as shown in Chapter 3. In case that the synchronization code is not separated from the functional part of the method inheritance enforces to break the encapsulation of the methods.

It is necessary to analyse, re-define, and re-implement the method get although the incremental modification of the buffer is the additional method get2 and not the modification of the behaviour of get. Depending on the implementation of the synchronization constraints it even can be necessary to re-implement all methods.

The described problem is not specific to object-oriented hardware design but is a general problem of designing object-oriented concurrent systems. There are some more scenarios similar to the one described above in which inheritance and synchronization constraints conflict with each other. In object-oriented concurrent programming these conflicts are called inheritance anomaly. A detailed description of inheritance anomaly is given in [10]. In [10] the anomaly is analysed, categorized, and illustrated by an example of a bounded buffer[1]. Three main categories of anomaly are described in [10]:

- Partitioning of acceptable states
- History-only sensitiveness of acceptable states
- Modification of acceptable states

The anomaly caused by the partitioning of acceptable states is the anomaly described above and illustrated with the derived buffer "x_buf2". The state "buffer_partial" in which the message to invoke the method get is accepted has to be partitioned into the states "buffer_partial_one" and "buffer_partial_more_than_one".

History-only sensitiveness means that the messages which are accepted only depend on the previous states of the object. For example, a buffer "gb_buf" derived from "bbuffer" has a method "gget" which behaves almost identical to the method get with the exception that it cannot be invoked immediately after the invocation of the method put.

The anomaly caused by the modification of acceptable states appears if new methods provoke the object to restrict the states in which it accepts the inherited methods. For example, the buffer "lb_buf" derived from "bbuffer" has two new methods "lock" and "unlock". After "lock" is executed no message to invoke a method inherited from "bbuffer" is accepted until the method unlock is executed.

Several attempts to solve the anomaly are described in [10]. Although only a solid solution is given to the state partitioning anomaly the technique to use guards for specifying the synchronisation constraints proposed by [7] and analysed in [10] appears to be a promising one. Therefore the solution of the inheritance anomaly presented in the next chapter is based on guards.

5. Solution of the inheritance anomaly

This chapter describes a modelling technique based on the object-oriented language extension to VHDL which solves the inheritance anomaly. In object-oriented programming several attempts have been made to solve the anomaly. A very interesting one was made in [7]. It is based on guards. In [7] a guard gives a condition under which a method cannot be accepted. For example "buf_in = buf_out -- buffer empty" could be a guard of the method

1This is the reason why the bounded buffer is used here.

get of the bounded buffer. However, it was shown in [10] that the guard mechanism presented in [7] does not solve every inheritance anomaly.

Another proposal to solve the anomaly by guarded methods is given in [6]. In this proposal methods are interpreted as nested conditional critical regions (CCRs). As a consequence, efficient implementation for CCRs have to be developed first, before the proposed concept can be implemented.

The solution presented in this chapter is also based on guards but avoids the drawbacks of the above proposals. One of the main ideas is to separate the protocol specification and implementation from the functional part of a method as already described in Chapter 3. Each method has an interface list which contains the object itself as a signal parameter and some other signals to communicate with other objects, e.g. clients. The principle structure of a subprogram implementing the protocol is composed as shown in Listing 5. Only if a message is detected to invoke

```
procedure protocol_part_method (
   signal this : inout taggedtype;
   signal message_in : in message'class;
   signal message_out : out another_message'class) is
begin
   if decode(message_in) = op_code then -- operation
decoded
      if guard_expression then
         method(this, message_in, message_out);
         additional protocol code;
      end if;
   end if;
end protocol_part_method;
```

Listing 5 Protocol implementation

the operation realized by the corresponding method and only if the guard expression is true then the subprogram is invoked which implements the functional part of the method. The internal call of the subprogram is dynamically bound,[1] i.e., for modifying the behaviour it is sufficient to derive a new tagged type which re-implements the subprogram. The inherited protocol then automatically calls the re-implemented subprogram. This all is done without consuming time. There must be no wait statements between the beginning of the subprogram implementing the protocol and the call of the method implementing the functional part.

Different to proposals from the software domain [6][7] a method guard is not part of the object-oriented language but is only part of the modelling technique to solve the anomaly. Any expression is allowed as a guard. If it is true it means that the message to invoke the method can be accepted. As simulation time does not proceed neither the message to invoke the method nor the guard expression can change between the invocation of the method implementing the protocol and the invocation of the method implementing the functional part of the method. If the message is not detected or the guard expression does not accept the method invocation then the protocol method also does not consume any time. The method which implements the functional part consumes time. It contains wait statements so that signal assignments in the method change the object's state and/or the output channel (message_out).

As an example the method get of the bounded buffer is modelled (see Listing 6).

```
procedure protocol_part_get(
   signal this: inout bbuffer;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channel : inout
          bufferreturnvalue'class) is
begin
   if decode(op_channel) = get then
      if not( this.buf_in = this.buf_out) then
         -- guard: buffer not empty
         execute_get(
            this => this, op_channel => op_channel,
            returnvalue_channel => returnvalue_channel);
     end if;
  end if;
end;
```

Listing 6 Protocol of method get

Both, "protocol_part_get" and "execute_get" are subprograms implementing a method of the bounded buffer. In the modelling technique all the methods have to be implemented in this style.

5.1. How to model a body of an object

The body of an object running in parallel to other objects is implemented as a concurrent procedure call which calls a subprogram "object_body". The procedure "object_body" consists of sequential procedure calls invoking all methods of the class (see Listing 7).

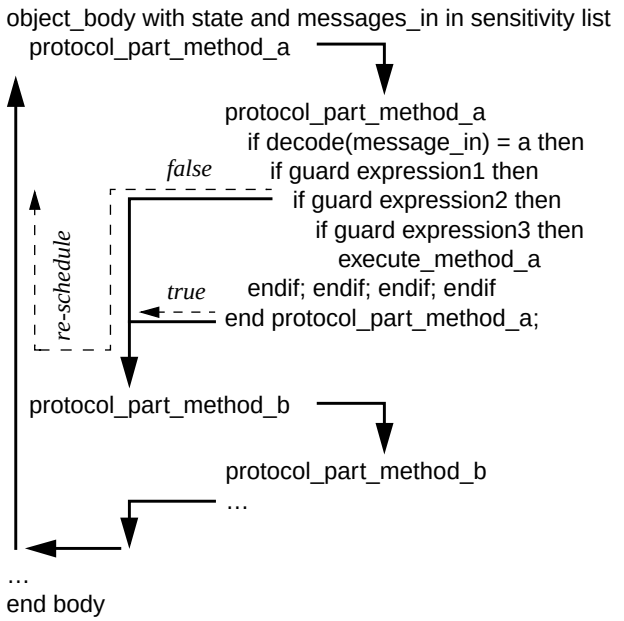
```
procedure object_body (
   signal this : inout bbuffer;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channel:out bufferreturnvalue'class)is
begin
   protocol_part_put(
      this, op_channel, returnvalue_channel);
   protocol_part_get(
      this, op_channel, returnvalue_channel);
end object_body;
```

Listing 7 Object body

The procedure starts when the state of the object changes or a message comes in. Then a method is invoked (the functional part) if the corresponding message appears

1 This is different to e.g. Ada where internal calls are statically bound by default

and the guard expression is true. If there are not any messages the objects waits for new messages. If there are messages but the corresponding guard expression is not true the method is not executed. The method is re-scheduled for the time when the object's state or any incoming message causes an event. Then it is tested if the message still is there and the *complete* guard expression is evaluated again. As it is shown later the guard expression can become more complex by inheritance. Nevertheless, it is completely evaluated when a method is re-scheduled (see



Listing 8 Re-schedule

Listing 8). As reported in [1] "the most difficult question is how to do something suitable with inheritance in the definition of the body (the local process of an object)". In some cases inheritance enforces to re-implement the complete body of an object. The problem is also described in [10] as *body anomaly* for objects which have a body with its own thread of control. Therefore the method implementing the object body has no thread of control beside the unavoidable sequential order of execution of the procedure calls. If a new method is added to a class a new body is defined. It contains a procedure call which invokes the protocol part of the new method. It also contains a sequential procedure call which invokes the old body. To denote the old "object_body" and to distinguish it from the overloaded procedure "object_body" the attribute 'Parent is used. This is illustrated for the example of the buffer "x_buf2" in Listing 9. That way the new object body remains compatible with the old one.

In the modelling style an instantiated object can be interpreted as a statemachine. The values of the object's attributes, i.e. record elements of the tagged record store the state of the object. Guard expressions containing values of the object's attributes distinguish different states of

```
procedure object_body (
   signal this : inout x_buf2;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channel : out
      bufferreturnvalue'class) is
begin
   protocol_part_get2(
      this, op_channel, returnvalue_channel);
   object_body'parent(
      this, op_channel, returnvalue_channel);
end object_body;
```

Listing 9 Object body of buffer x_buf2

the object or in other words, they determine equivalent states with respect to the possibility to invoke methods. A method can only be invoked by a message if the object is in a state allowing the method to be executed. As described above the states are determined by the guards of a method. If a method is executed the object performs a transition in its state space. The next state depends on the values which are written into the attributes of the objects by the method.

The guard expression for the method get of a buffer of distinguishes between two states: "empty" and "not empty". The guard expression of put distinguishes between two other states: "full" and "not full". The resulting state space consists of three states: "full", "partial", and "empty".

5.2. Partitioning of acceptable states

The anomaly caused by the partitioning of acceptable states can be solved by the guard mechanism. The new method which requires the partitioning of the states just has a new guard expression which partitions the state. It is illustrated by the example "x_buf2" (compare Chapter 3) in Listing 10.

The guard which checks if there are at least two items in the buffer partitions the state "partial". Each of the new resulting states behaves the same as the old state "partial" with respect to the inherited methods put and get.

If states are partitioned into new states by adding new attributes to the class it might be necessary to refine existing transitions. It might be necessary to state more precisely which partition of the old state becomes the target of the new state on which condition. This requires to add protocol code to existing methods in form of synchronization constraints (guard expressions) and/or assignment statements which assign values to the new attributes. This has to be done without breaking the encapsulation and while staying compatible with the existing protocol. The proposed solution in the modelling technique is to embed the existing functional part of the method in a re-defined method which overloads the method implementing the functional part. The re-defined method contains the additional protocol information. As described above, the implementation of the synchronization constraints, i.e. the

```
x_buf2 is new bbuffer with null; end record;

procedure protocol_part_get2(
   signal this: inout x_buf2;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channe : inout
      bufferreturnvalue'class) is
begin
   if decode(op_channel) = get2 then
      if not(this.buf_in = this.buf_out) and
         -- guard: buffer contains more than one element
         not( (this.buf_out + 1) mod (this.buf'length + 1)
         = this.buf_in) then
         execute_get2(
            this=>this, op_channel => op_channel,
            returnvalue_channel => returnvalue_channel);
     end if;
  end if;
end;
```

Listing 10 x_buf2

guard expressions must not consume any time (see Listing 11). Although new guard expressions are added to the pro-

```
procedure protocol_part_method
begin
  if decode(message_in) = op then
     if guard expression then
           execute_method ──────────────┐
                                        ▼
                          procedure execute_method
 no wait statements       begin
 allowed                     if guard expression2 then
                                        execute_method'parent
                             endif;
                             additional protocol code;
                          end execute_method;
                ▲
                │
      endif;
   endif;
additional protocol code
end protocol_part_method;
```

Listing 11 Refinement of protocol code

tocol code the re-schedule mechanism as shown in Listing 8 still works.

5.3. History-only sensitiveness of acceptable states

The described modelling style allows to solve the anomaly caused by the history-only sensitiveness of acceptable states. It is only necessary to introduce an attribute which traces the history, to partition the states, and to refine the transitions as described above. As an example the buffer “gb_buf” (compare Chapter 4) is shown in Listing 12.

```
type gb_buf is new bbuffer with
  after_put: boolean; -- traces invocation of put
end record;

procedure protocol_part_gget(
   signal this : inout gb_buf;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channel : inout
      bufferreturnvalue'class) is
begin
   if decode(op_channel) = gget then
      if not( this.buf_in = this.buf_out)
      and after_put = false then
         execute_gget(
            this=>this, op_channel => op_channel,
            returnvalue_channel => returnvalue_channel);
   end if; end if;
end;

procedure execute_get(  -- OVERLOADING !!!
   signal this : inout gb_buf;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channel : inout
      bufferreturnvalue'class) is
begin
   -- always: guard = true
      execute_get'parent(
         this=>this, op_channel => op_channel,
         returnvalue_channel => returnvalue_channel);
   this.after_put <= 'false'; -- refinement of transition
   wait on clk = '1'; -- or wait for 0 ns; ect.
end;

procedure execute_put(   -- OVERLOADING !!!
   signal this : inout gb_buf;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channel : inout
      bufferreturnvalue'class) is
begin
   -- always: guard = true
      execute_put'parent(
         this=>this, op_channel => op_channel,
         returnvalue_channel => returnvalue_channel);
   this.after_put <= 'true'; -- refine transition
   wait on clk = '1'; -- or wait for 0 ns; ect.
end;
```

Listing 12 gb_buf

In the buffer “gb_buf” only the value of the additional attribute is specified for each method to select the new transitions.

5.4. Acceptable states anomaly

The refinement of conditions which enable transitions by putting additional guard expressions to existing methods allows to solve the modification of acceptable states

anomaly. The modification can be seen as a refinement of the transition conditions. Therefore the modification can be expressed by an additional guard. This is illustrated by the example of the buffer “lb_buf” (compare Chapter 4) in Listing 13.

```
lb_buf is new bbuffer with
   lock: boolean;
end record;
```

Listing 13 lb_buf

Methods to lock and unlock the objects are added to the class. A new guard expression which checks if the object is locked or not is added to each method (see Listing 14).

```
procedure execute_put(  -- OVERLOADING !!!
   signal this: inout lb_buf;
   signal op_channel : in bufferoperation'class;
   signal returnvalue_channel : inout
      bufferreturnvalue'class) is
begin
   if this.lock = false then
      execute_put'parent(
         this => this, op_channel => op_channel,
         returnvalue_channel => returnvalue_channel);
   end if;
end;
```

Listing 14 New guard expression in method put

Again it can be seen that it is not necessary to break the encapsulation of the object or more precisely of the methods to solve the anomaly. It is important to note that the protocol part only can be incrementally modified by adding new synchronization constraints. It is not possible to remove a guard expression. In such a case the protocol would not be compatible any more (compare Chapter 3).

The concept to solve the anomaly is compatible to genericity extensions like e.g., the one described in [2]. Especially, mixin inheritance can be used without worrying about acceptable states anomaly.

In summary one can say that the object-oriented language extension to VHDL in combination with the presented modelling technique can be used to solve all the inheritance anomalies without breaking the encapsulation of the class. It is neither necessary to analyse existing protocol code or functional code nor to re-implement them when inheriting methods from a parent class.

6. Conclusion

This paper has analysed the inheritance anomaly with special respect to object-oriented hardware design. A modelling technique was presented which is based on an object-oriented language extension to VHDL. In this context it was shown that event driven simulation allows the modelling of guards as appropriate synchronisation constraints in concurrent systems.

In summary, one can say that there is a solution to the inheritance anomaly in object-oriented hardware design.

7. References

[1] America, P.: Inheritance and Subtyping in a Parallel Object-Oriented Language. in Bézivin, J.; Hullot, J-M.; Lieberman, H. (eds.): European Conference on Object-Oriented Programming ECOOP ’87, Lecture Notes in Computer Science 276, Springer 1987

[2] Ashenden, P., J.; Wilsey, P., A.; Martin, D.E.: Reuse Through Genericity in SUAVE. in Rapid System Prototyping with VHDL. Proceedings of the VIUF Fall 1997 Conference, 1997

[3] Ashenden, P., J.; Wilsey, P., A.; Martin, D.E.: SUAVE: Painless Extension for an Object-Oriented VHDL. in Rapid System Prototyping with VHDL. Proceedings of the VIUF Fall 1997 Conference, 1997

[4] Benzakki, J.; Djafri, B.: Object Oriented Extensions to VHDL–The LaMI proposal. CHDL’97, Toledo, Spain, 1997

[5] Cabanis, D.; Medhat, S.: Object-Oriented Extensions to VHDL: The Classification Orientation. Proceedings of the VHDL User Forum Europe 1996, Shaker Verlag 1996

[6] Ferenczi, S.: Guarded Methods vs. Inheritance Anomaly Inheritance Anomaly Solved by Nested Guarded Method Calls. ACM SIGPLAN Notices, Volume 30, Number 2, Februar 1995

[7] Frølund, S.: Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. in Lehrmann Madsen, O. (ed.): European Conference on Object-Oriented Programming ECOOP ’92, Lecture Notes in Computer Science 615, Springer 1992

[8] Glunz, W.; Pyttel, A.;Venzl, G.: System-Level Synthesis. in Michel P.; Lauther U.; Duzy P. (eds): The Synthesis Approach to Digital System Design. Kluwer Academic Publishers, p. 221-260 , 1992

[9] IEEE Standard VHDL Language Reference Manual Std 1076-1993, Revision of IEEE Std 1076-1987, 1994

[10] Matsuoka, S.; Yonezawa, A.: Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. in Agha, G.; Wegner, P.; Yonezawa, A. (eds.): Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993

[11] Perry, D.: Applying Object Oriented Techniques to VHDL. Proceedings of the VIUF Spring Conference, p. 217-224, 1992

[12] Radetzki, M.; Putzke-Röming, W.; Nebel, W.: Language architecture document on Objective VHDL. REQUEST Report D1.2C, ESPRIT Project 20616, OFFIS, LEDA, France Télécom, Italtel, 1996

[13] Schumacher, G.; Nebel, W.: Inheritance Concept for Signals in Object-Oriented Extensions to VHDL. Proceedings of the EURO-DAC ’95 with EURO-VHDL ’95. IEEE Computer Society Press, 1995

[14] Swamy, S.; Molin, A.; Covnot B., M.: OO-VHDL Extensions to VHDL. Computer, October 1995 pp. 18–26, IEEE, 1995

[15] Wegner, P.; Zdonik, S. B.: Inheritance as an Incremental Modification Mechanism or What Like Is and Isn‘t Like. in Gjessing, S.; Nygaard, K. (eds.): European Conference on Object-Oriented Programming ECOOP ’88, Lecture Notes in Computer Science 322, Springer 1988