# Inheritance Concept for Signals in Object-Oriented Extensions to VHDL

Guido Schumacher, Wolfgang Nebel
FB10 - Department of Computer Science
Carl von Ossietzky University Oldenburg
Ammerländer Heerstraße 114 - 118
D-26129 Oldenburg

EMAIL: guido.schumacher@informatik.uni-oldenburg.de

**Abstract**

*Several proposals were made in the last few years to extend the hardware description language VHDL and to add mechanisms like inheritance from the object oriented domain to the language. This paper illuminates the principle problems arising when an inheritance concept for data types is added to VHDL. Solutions to these problems are proposed with an example of an inheritance mechanism for signals within an object-oriented extension to VHDL.*

## 1 Motivation

With the increasing use of object-oriented techniques in software engineering a lot of methods from this area were investigated by hardware-designers. Some of these techniques seem to be appropriate to solve problems in the hardware domain which are similar to the ones from the software domain. This has led to several proposals on how to adapt these techniques in hardware design [4][7][8][10] [13][17]. One particular way of introducing object-oriented techniques to hardware design is to develop object-oriented hardware description languages. A common method of doing this is to extend an existing language to an object oriented one. On the one hand it is possible to extend an existing object-oriented programming language to a hardware description language [19] on the other one can extend an hardware description language to an object-orientated one. There exist quite a few proposals for extensions especially for VHDL [1][2][3][6][15][16]. Section 3 gives an overview of existing proposals for extensions. Most of them [1][2][3][6][15] introduce inheritance-mechanisms for data types or use data types to realize inheritance mechanisms on object oriented classes. When applying these mechanisms some problems have to be considered. Section 4 lists the difficulties. To overcome the problems an inheritance concept for signals is proposed in section 5. Section 6 illustrates the modelling technique based on inheritance of signals with an example. Some conclusions are drawn for further steps of extending VHDL.

## 2 Objectives and requirements of object-oriented hardware-design

A high level of abstraction becomes a more and more important objective in the hardware system specification. A list of requirements for this objective is given in [5]. Mainly the higher complexity of designs, the better readability of descriptions, the abstract description as a key to faster simulation and the impossibility to manage a design at low level of abstraction are mentioned. To achieve the goal of a higher level of abstraction a view is proposed in [5] in which a processor offers well defined services. These services can be used by other processors by the invocation of operations. The processors are modelled as abstract data types. From this point of view the later hardware realization is an implementation of an abstract data type. In object-oriented terminology these data types correspond to classes and the services/operations to methods.

To achieve a high level of abstraction simple techniques for an invocation of a method have to be provided by the specification tool. At the same time these tech-

niques should be flexible enough to be adapted to different system requirements which may also change during the design process. The methods of an object should be available to several other objects.

In order to design efficiently, the reuse of hardware-descriptions is an objective in system specification. An appropriate approach is the use of an inheritance concept. Existing objects must be expandable by adding properties to them and by modifying the methods.

The full power of a an inheritance concept can be obtained when polymorphism is provided for the methods. It is possible to have several versions of a method according to the inheritance hierarchy. In a concept of polymorphism the identification of the modifications of an object and the selection of the corresponding method is done automatically during runtime. This is different from static subprogram and operation overloading. The advantage of polymorphism is that a heterogeneous object-container can contain objects of different classes during runtime which are derived from each other. All their methods can be invoked in the same way. The designer doesn‘t have to distinguish different cases which may appear during invocation depending on the different objects‘ classes, especially he doesn‘t need to know anything about future derivations of the objects.

An inheritance mechanism is also appropriate for refinement during the synthesis process. Objects can be refined by using an inheritance mechanism.

To reach the objectives of the object-oriented hardware-design not only during the specification phase but during the whole system development process a link from the object-oriented methods to further synthesis steps is necessary. It is important that the realization of the object-oriented techniques is feasible for hardware.

The difficulties of these requirements are discussed in section 4 with special respect to solutions using object-oriented extensions to VHDL.

## 3 Object-oriented extensions to VHDL

This section gives a short overview of proposed object-oriented extensions to VHDL. In [15] a distinction between object-oriented support to the structural hierarchy and object-orientation of data-types is suggested. Another classification one can think of is the distinction between extensions which are appropriate for synthesis and extensions which are appropriate only for simulation.

A proposal which introduces a class concept into VHDL is given in [1]. The class concept is derived from the class concept in C++. A class[1] consists of attributes and methods like in C++. A translation mechanism from the extension called VHDL++ to VHDL is introduced. For the instantiation of classes VHDL global signals are used. The methods become procedures with the global signals as parameters after the translation. Methods containing a write access to an attribute have to be invoked within one process only because no resolution function is introduced in the translation process. As the methods are translated into normal VHDL procedures accessing global signals an overloading mechanism during runtime is not provided. The straight forward translation mechanism enables to use the translated VHDL-code for synthesis. This approach is in-between the support to the structural hierarchy and object-orientation of data-types. On the one hand the data-type concept of VHDL is not changed on the other no hierarchy concept for the new class construct is introduced.

Another class approach based on the C++ class concept is presented in [3]. The extension is called VHDL_OBJ. A class-construct contains attributes and methods to access the attributes. Like in C++ the methods can be declared virtual. This indicates that it is a polymorphic method and that it can be overloaded during runtime. The implementation of VHDL_OBJ is a translation of the new language constructs into VHDL. Attributes are translated into records and methods are translated into procedures. To distinguish these polymorphic procedures they get as a first parameter a pointer (access type) to the record related to the attributes of the classes. If an object of a polymorphic class is instantiated a record of pointers is allocated. One pointer points to the actual object. Two limitations due to the use of access types exist. Class-objects cannot be shared by different processes and they are not appropriate for synthesis. Again like in the previous approach the data-type concept of VHDL isn‘t changed and no new concept for the hierarchical structure of a VHDL model is introduced. Class becomes a new language element which stands amongst the others.

The revision of the IEEE VHDL language standard [18] introduced the concept of shared variables. Object-oriented-programming is mentioned in [20] as reasonable use of shared variables. Based on this idea it is suggested in [15] to use shared variables as objects in the object-oriented meaning. To implement the methods a monitor concept is proposed. The monitor schedules the access to the shared variables by procedures which can be seen as primitive operations. It is additionally suggested to introduce an inheritance concept and a late binding concept to VHDL. As shared variables were introduced in VHDL for modelling at system level and as they do not have an equivalent in existing hardware they are not appropriate for synthesis.

1. The terms class and attribute are utilized in this paper as they are used in the object-oriented world. It is marked if they are used in the VHDL-sense

Similar to VHDL a revision of the IEEE standard of Ada has to take place [12]. Mainly object-oriented features are added to the language. A concept to derive types is provided by the introduction of tagged types. Data encapsulation can be done by records. Marking records as tagged makes these records expandable. Other record types can be derived from the tagged records.The attribute ‘CLASS defines an unconstrained type a so called class-wide-type which is the union of all types derived from a base type. A procedure call with a parameter of class-wide-type invokes a dispatching mechanism for this procedure. To use the same constructs in VHDL is proposed in [6]. It is stated in [14] that this is a more evolutionary process in Ada than this would be in VHDL. In addition to the inheritance mechanisms for data types inheritance mechanisms for entities and architectures are part of the proposal to extend VHDL in an Ada-like way. A concept for synthesis is not considered within the proposal.

The message-passing mechanism as suggested in [16] realizes communication not only by signals, but also by message-passing between EntityObjects. On receiving a message the EntityObject invokes a method. The destination of a message can be determined during runtime by a so called handle which is associated with an EntityObject. The handle is some kind of address of an EntityObject. If several methods of an entity or EntityObject are invoked in the same simulation cycle then a queuing-mechanism stores the invocations for sequential execution. An inheritance mechanism allows to derive new EntityObjects from existing ones. Furthermore constructs from Ada are used to add control functionality for concurrency. A preprocessor is proposed for translating the new language constructs into VHDL. As a consequence the whole message-passing mechanism is translated into a fixed VHDL algorithm. As a result an overhead is produced in cases where a queuing mechanism isn‘t necessary. At the same time a communication mechanism is used which can‘t be refined and changed in further synthesis steps. Hence this approach is not intended for synthesis but for rapid prototyping. In [15] the proposal is classified as object-oriented support to the structural hierarchy.

A conclusion can be drawn that for quick modelling in an object-oriented way at a high level of abstraction the cited approaches offer similar possibilities. What still has to be compared and analysed are the inheritance concepts and the links to further design steps.

# 4 Problems with inheritance concepts

This chapter mentions problems which have to be solved when introducing an inheritance concept in hardware specification.

## 4.1 Invoking a method in hardware

Object-oriented methods are all based on the idea that there are different objects interacting with each other by methods related to them. Classifying objects with the same properties leads to the term class in the object-oriented domain. If we send a message to an object in software this corresponds to a procedure or function call. Return values are given back by simple assignment statements. It‘s a bit more complicated in hardware-design. On condition that an object has its counterpart in an existing piece of hardware, a protocol mechanism has to be started to invoke a method of such an object. Similar a protocol has to be used to handle the return values.

## 4.2 Fix protocol mechanisms as part of the specification language

The cited approaches offer fixed protocol mechanisms for invoking methods and getting return values as part of the language. This has some consequences. The transportation of the messages has to be realized by signals with the exception of the shared-variable-approach. This has to be done even if the signals aren‘t visible to the designer. In that case additional delta cycles are introduced into the simulation to run the protocol. In contrary to protocols running on existing hardware the simulated protocols don’t consume any (simulation-) time. It is not provided for further refinement and synthesis steps to replace the protocols by more appropriate and accurate protocol models which consider the timing behaviour. This might be important e.g. for simulating the specification together with already synthesized parts of a system. It is also not provided to do a synthesis on the protocol itself although protocol generation needs some attention in a design process [7][21]. The protocols in the proposals are developed for high-level-specification but do not fit in most cases for real hardware. In some cases this leads to an overspecification.

## 4.3 Polymorphism in hardware

Another problem is the realization of polymorphism in hardware. In case a method is a polymorphic procedure in software, the addressing of the object is realized by pointers. The analogous use of pointers in a polymorphism-concept in a hardware-description-language means that it is only appropriate for simulation. The introduction of handles to address the different objects like in [16] is a way to allow polymorphism in a hardware-specification. But it has some strong restrictions. In [16] it is only mentioned that handles can be stored in variables. So it is not possible to pass them between different processes.

### 4.4 Construction of new classes

Construction of new classes from the existing ones not only by composition of attribute classes as mentioned in section 3 but also by inheritance is the appropriate technique for efficient code-reuse. Although in principle performing the same extension-mechanism, the constructs for the inheritance-mechanisms are very specific in the different proposals. The proposals introduce new language constructs which aren‘t based on constructs already available in the language. Additional attributes of an inherited EntityObject are introduced by extra instance variables of a specific type, additional attributes in a shared variables monitor are brought in by additional shared variables of a specific type. Extra methods can be related to the new set of instance variables or shared variables.

To introduce an inheritance mechanism which is based on constructs already in the language, it is only necessary to combine the set of variables (instance-variables or shared variables) in a record and apply the inheritance mechanism on the type of that record. This inheritance mechanism is simple and even could be used by EntityObjects or shared variables monitors e.g. it is a general mechanism for inheritance. It is even compatible to the cited approaches. The methods then are no longer related to a set of variables but to a type of a record representing the variables.

### 4.5 Inheritance mechanism based on records

To base the inheritance mechanism on records is already proposed in [6]. But in that approach problems are still remaining. A link to further design steps is not provided by the proposal, it‘s only intended to add to the abstractness of VHDL and make VHDL designs more easily modifiable and reusable. The second problem is that class-wide-types as mentioned in section 3 are introduced as unconstrained types. This is because it can‘t possibly be known how much space could be required by any value of a class-wide-type. The type might be extended after the compilation of the design-unit containing the class-wide-type. As a consequence, although an object can be declared as a class-wide-type it must be initialized and it is then constrained by the type of the initialization value. Therefore an heterogeneous object-container carrying various objects of different classes can‘t be realized using signals. These heterogeneous object containers serve as a basis for polymorphism. Because of that, no signals but only variables of access-type can be used for polymorphism. Since signals are the only possibility to exchange data between components and processes the class-wide-types as unconstrained type is a severe restriction.

To give up the restriction and allow class-wide-types to be a constrained means to provide a general powerful inheritance mechanism to VHDL. In that case a constrained object of a class-wide-type has to provide enough ‘memory‘ to be able to contain every object of a derivated type. To analyse that ‘memory‘- requirements all previous compiled code has to be re-analysed. As a result heterogeneous object containers could be modelled by using signals.

At the same time constrained class-wide-types would be a solution to make protocol specifications flexible and expandable. As mentioned above it is often necessary to adapt protocols to specific purposes. The connection between processes and components described by protocols are realized by signals. The state of a protocol is stored in variables which correspond to registers and the transitions between the states can be described by procedures. If signals, variables and the associated procedures can easily be extended by the proposed inheritance mechanism then the protocol itself can be extended.

## 5 General inheritance concept on data types

In this section an inheritance concept for record data types is presented. It allows to re-use and extend a type-definition for records. It is combined with a mechanism for polymorphism. In order to present and discuss the inheritance concept, a syntax will be used which is an extension to VHDL‘93 [18]. The syntax of the language extension is based on the syntax for tagged types in Ada9X and therefore similar to the one proposed in [6]. However the semantic is different. Secondly a mapping to VHDL is presented.

### 5.1 Language extension

In a record type definition it is possible to mark the record as expandable by the key-word **tagged**. Using the keyword **new** a new record can be derived in another type-definition from the tagged one by referencing the tagged one. The new record inherits all elements of the old one. It then consists of the elements of the old record and the ones given in the record_type_definition. To allow the specification of abstract classes it is possible to declare an empty record. This is done by the keyword **null** instead of an element declaration. The syntax for the new tagged record type definition is as follows:

```
tagged_record_type_definition ::=
    tagged record
        element_declaration {element_declaration}
        | null ;
```

**end tagged record**

| **new** *type*_name **with** record_type_definition

| **new** *type*_name **with record null end record**

composite_type_definition ::=

array_type_definition

|record_type_definition

|tagged_record_type_definition

For each tagged record type definition a corresponding class wide type exists which is the union of all types derived from that tagged record type. This type is by definition a constrained type. By attaching the attribute **'CLASS** to a type name of a tagged type the class wide type is referenced. VHDL-objects of arbitrary types can be assigned to any VHDL-object of a class-wide types as long as these types are derived from the tagged type corresponding to the class wide type. This is different from the concept proposed in [6].

As a consequence it is possible to model connections between processes and components with signals of class wide types. This allows a designer to model communication mechanisms in an early phase of the design without a knowledge of the final solution for this mechanism, because he is able to evolve it and easily expand it during the design process. For example a connection of a class wide type transports instructions from one component to another. If during the design process a new type of instruction occurs which needs a more sophisticated method to transport it, because it has additional arguments, then this can be included without modifying the transport mechanisms of the other instructions.

Although it is one of the main advantages of this concept that it allows the modelling of interprocess communication in an object-oriented way it is not limited to signals. It can be used for variables too. An analogy can be seen between the extension of tagged types and the specialisation of subtypes. In the same way constraints are added to subtypes, extensions are added to tagged types. However, a difference is that subtypes are compatible whereas tagged types derived from each other are not. The compatibility has to be declared explicitly by the use of the attribute 'CLASS.

Along with the possibility to construct new classes (types) by derivation of existing ones the composition of attribute-classes as indissoluble subobjects is possible. The encapsulation concept doesn't need to be changed for the extension.

A powerful mechanism which operates on the incompatibility of the derived types is polymorphism. It is part of the presented inheritance concept on data types. The polymorphism-mechanism runs on class-wide types. When calling a function or procedure with a class-wide type as an actual of mode in or inout then the type of the actual is determined during runtime and the corresponding function or procedure is invoked. In contrast, the VHDL operator overloading is done statically during elaboration.

The possibility of modelling methods especially for communication mechanisms with respect to their timing behaviour can cause some problems when using the dispatching mechanism. It is a problem if a method (procedure/function) is invoked and the invoking class changes before the result of the method call is available or rather before the method call is completely executed, i.e. the type of the actual being a signal and starting the dispatching mechanism has changed before the function or procedure has reached its end. In general this problem can arise in language extensions of VHDL if they allow to declare heterogeneous object-containers with methods invoking a dispatching mechanism and consuming time. This can happen if the content of the object-container can change in parallel to the execution of the method, e.g. when they run in different not synchronized processes. The following example together with figure 1 illustrates the situation. A

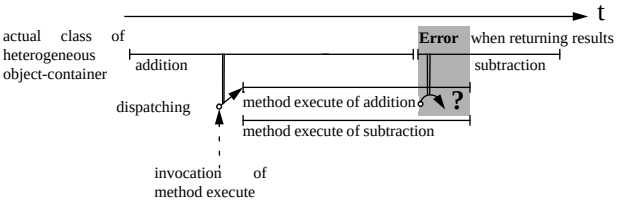


**Fig. 1 Change of class during method-execution.**

class 'addition' and a class 'subtraction' are both derived from the abstract class 'instruction'. An object of a class wide type instruction'CLASS is used as a heterogeneous object-container to model an instruction register. The actual class of the object is 'addition'. The method 'execute' of the class 'instruction' is invoked. Before the addition has finished the content of the object changes and the actual class now is 'subtraction'. In the proposal for the language extension such a situation is a run-time error.

## 5.2 Mapping to existing hardware description language

To embed the inheritance concept on data types in a design flow, a link to existing design environments is necessary. For the presented language extension this link could be a translation into VHDL. In the following such a translation-mechanism is presented.

The derivation and inheritance is mainly done by copying the record elements and the methods. A class-

wide type is translated into a record which contains all elements of the corresponding tagged type and records derived from it. Some kind of variant record management could be introduced as an improvement. A so called tag is assigned to each tagged record. Corresponding to the inheritance hierarchy of the tagged type a subtype hierarchy of an enumeration type is created. After translation, each record of a class wide type gets an additional element of the corresponding enumeration type containing the actual type (tag) during runtime.

Checks or assignments of the tags during runtime have to be done in assignment statements containing a class-wide type.

In case of overloading a subprogram with an actual of mode in/inout of a class wide type it has to be analysed if the actual subprogram can be determined during compile time. If not, a procedure or function call has to be replaced by a procedure or function call to a subprogram containing a dispatching mechanism. This mechanism consists of a simple case statement, in which the choices are the different possible tags of the class-wide type. The possible tags depend on the visibility of the corresponding type at the location of the subprogram call. The sequence of statements in the case statement alternative is a call to the procedure/function attached to the corresponding type.

If the object of the class wide type is a signal then the interface of the subprogram has to be modified for that call. An additional parameter of the enumeration type corresponding to the tag of the case statement alternative is introduced into the interface. This parameter is associated to the tag-element of the record of the class wide type. It is introduced to detect a change of the tag during the execution of the subprogram. In order to lower the risk of type errors in the dispatching mechanism during runtime it is only allowed to declare a subprogram and the corresponding tagged type together in one package. This avoids the situation that the scope of the tagged type declaration is different from the scope of the subprogram declaration. In other words it‘s not possible to have a choice in a dispatching mechanism without the corresponding subprogram being visible in the case statement.

On the one hand the suggested translation has some obvious disadvantages. It isn‘t trivial, e.g. all derived types of a tagged type have to be known to the translator before it can construct the corresponding class wide type. The implementation of a variant record management needs sophisticated algorithms.

On the other hand the presented inheritance concepts enables the designer to solve the problems mentioned in section 4. It allows to model the invocation of methods in hardware in an object-oriented way. It overcomes the problem of fix protocol mechanisms. The protocol mechanism to invoke methods in (hardware-) objects is no longer part of the language but nevertheless it can be (re-) used as easy as if it were a part of the language. A small example is presented in the next section to illustrate this. A polymorphism concept has been introduced and by including signals there are no limitations concerning the exchange of handles between processes. By applying the inheritance mechanism to records which are already part of the language, no artificial additional constructs have to be used to model objects and introduce an inheritance concept.

## 6 Example

The following section gives an example on how to use the proposed inheritance concept for signals to specify an expandable model. A counter is specified as an entity and architecture which can execute instructions like increment or get-value. The interface of the counter consists of two signals. A protocol is defined on those signals for the data exchange. The protocol is defined in a very general way so that it can be used not only by the counter but also by other design entities for communication. Hence the protocol could be defined in a library. At the same time it is expandable and can be easily modified and adapted to e.g. other design situations or further synthesis steps because it is defined in an object-oriented manner as a tagged type and a polymorphic procedure. The structure of the counter using the protocol is shown in principle in figure 2. A full description of the example including an

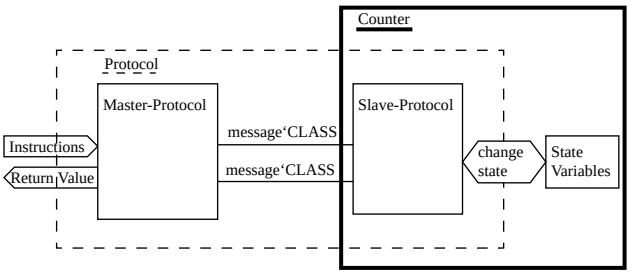


**Fig. 2 Example counter.**

overview of the inheritance hierarchy and the relations between the objects in a simplified version of the notation of Shlaer and Mellor [11] can be obtained from the authors by email.

There is a master-protocol which can be executed from outside the counter. The instructions of the counter can be passed as parameters to the master-protocol. The protocol transports the instructions via the message line to the slave-protocol which runs inside the counter. The slave protocol then starts the execution of the instruction. The instruction changes the state of the counter by changing its state variable. A return value is sent back by the slave-pro-

tocol to the master-protocol. The master-protocol returns the value to its caller.

Both parts of the protocol, the master and the slave protocol are related to a tagged type `message` as polymorphic procedures. These protocol operations consist of four of the five operations mentioned in [21] as atomic operations of a protocol. The waiting for a fixed time interval as an atomic operation is omitted because the counter model doesn‘t consider the timing behaviour in the first specification step. It can be implemented in another protocol related to a derived type `message_new`.

The instructions of the counter are represented by data types which are derived from a type `operation`. The type `operation` itself is derived from a tagged type `datum`. The type `message` has the class wide type `datum‘CLASS` as an record element i.e. the class `message` has an indissoluble subobject of the class `datum‘CLASS`. Hence the master-protocol can transport the instructions via the message line to the slave-protocol as mentioned above. The slave-protocol inside the counter reacts on such a message consisting of an instruction by calling a procedure to change the state of the counter. This is done by invoking a polymorphic procedure `exec_operation`. Depending on the actual instruction or more precisely the type of the instruction during runtime the related procedure `exec_operation` is invoked. This procedure has access to the state variable of the counter. It returns the result of the instruction to the slave-protocol which sends it to the master-protocol. The master-protocol then returns the value to its caller. This is illustrated in figure 3. Using this mechanism there is no

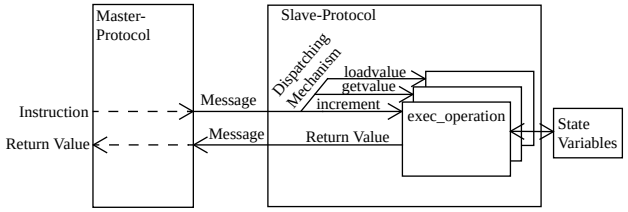


**Fig. 3 Protocol and dispatching of counter.**

problem to extend the model of the counter and to introduce new instructions. Only a new type derived from the type `operation` has to be defined. A procedure `exec_operation` has to be declared which changes the state of the counter as intended by the new instruction. In the counter example this has been done for an instruction `loadcounter`. Arguments of the instruction like the load-value are defined as elements of the record i.e. as indissoluble subobject of the class `loadcounter`. The entity or the architecture of the counter hasn‘t to be changed for that extension.

The example of the counter shows how it is possible to construct expandable and therefore re-usable objects by using the (predefined) types `datum`, `message` and `operation`. These types and the related procedures are so general that it is possible to model arbitrary objects. To model a new object only a set of operations has to be defined. The modelling style which appears cumbersome at first glance turns out to be simple and appropriate for reuse.

Not only the function of the counter but also the protocol mechanism can be extended by deriving a new type `message_new` from `message.`

Another example on how to use the dispatching mechanism could be the application in resolution functions. It would allow to flexibly specify a bus connecting components running different protocols. New components with new protocols could be added to the existing ones without considering them when specifying the first components and protocols. Figure 4 shows such a situation. If

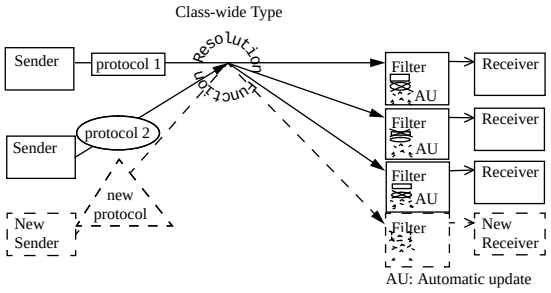


**Fig. 4 Protocol merging with resolution function.**

new senders and receivers are introduced it is sufficient to specify the resolution mechanism between the new and the old protocols in a function looking like this:

```
function resolved2(p:newprotocol;
   q:oldprotocol1) return protocol‘CLASS;
function resolved2(p:newprotocol;
   q:oldprotocol2) return protocol‘CLASS; …
function resolved2(p:newprotocol;
   q:oldprotocoln) return protocol‘CLASS;
```

Additionally the resolution between conflicts of the new protocol has to be defined.

```
function resolved2(p:newprotocol;
   q:newprotocol) return protocol‘CLASS;
```

The resolution function of the class wide type consists of repeated calls of the polymorphic procedures resolved2. It looks like the following function:

```
function resolved ( s: protocol_vector)
   return protocol‘CLASS is
begin
   if s‘LENGTH > 1 then
```

```
    return  resolved2(s(0),
              resolved(s(1 to s'HIGH))
            );
  else
    return s(0);
  end if;
end;
```

This procedure like the other ones to solve the conflicts between the old protocol types needn‘t to be changed if the new protocol is introduced, this is automatically done by the dispatching-mechanism.

The filter in figure 4 is a polymorphic procedure. It passes only the protocols which can be understood by the receiver. Introducing a new protocol means that the functionality of the old filters is updated automatically in the dispatching mechanism. Hence the system can be expanded only considering the new features the old parts doesn‘t have to be adapted or re-written.

## 7 Conclusion

Inheritance concepts for signals have been discussed. An object-oriented extension to VHDL has been presented which introduces an inheritance mechanism based on data types. Together with the inheritance mechanism polymorphic procedures on these data types are presented. The important point of the presented concept is that it is applicable to signals. It was shown that the inheritance concept especially for signals can be used to model a flexible and expandable protocol mechanism for the communication process between components in an object-oriented way. In contrast to other cited proposals high-level communication mechanisms are not part of the proposed language extension. Nevertheless such constructs can be introduced by derivable data types. Because the communication doesn‘t consist of a fixed structure but is based on an expandable concept, a good link to further design steps is provided. Additionally an idea on how to translate the proposed language extension into VHDL has been given. Problems arising when a dispatching mechanism is used for modelling a communication mechanism with respect to its timing behaviour have been discussed. An example has been presented to give an impression of how to use the proposed language extensions for object-oriented hardware-design.

## 8 References


[1] Glunz, W.: Extensions from VHDL to VHDL++. Siemens AG, ZFE BT SE 61 , 1991 , JESSI-AC/S2-WP1-T2.4-Q3

[2] 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

[3] Zippelius, R.; Müller-Glaser, K. D.: An Object-oriented Extension of VHDL. Proceedings of the VHDL-Forum Spring‘92 Meeting, 1992

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

[5] Oczko, A.: Hardware Design with VHDL at a Very High Level of Abstraction. Proceedings of the 1st European Conference on VHDL Methods, 1990

[6] Mills, M. T., Lt. Col.: Proposed Object Oriented Programming (OOP) Enhancements to the Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL). Final Report for 05/04/92-08/04/93 Solid State Electronics Directorate Wright Laboratory Air Force Materiel Command Wright-Patterson Air Force Base, Ohio 45433-7331

[7] Müller, W.; Rammig, F. J.: OO-Hardwaredesign ODICE: Object-Oriented Hardware Description in CAD Environment. Proceedings of the IFIP Ninth International Symposium on Computer Hardware Description Languages and their Applications, p. 19-34 , 1990

[8] Verschueren, C.: An Object Oriented Design and Simulation System for VLSI. Microprocessing and Microprogramming 30, p. 241-246

[9] Wayne, H. W.: How to Build a Hardware Description and Measurement System on an Object-Oriented Programming Language. IEEE Transactions on Computer Aided Design, March 1989 p. 288-301, 1989

[10] Glunz, W.; Venzl, G.: Harware-Design using Case Tools. Proceedings of IFIP VLSI‘91, 1991

[11] Shlaer, S.; Mellor, S. J.: Object-oriented system analysis: modeling the world in data. Yourdon Press computing series, 1988

[12] ISO/IEC JTC1/SC22 WG9 N 193: Programming Language Ada, Language and Standard Libraries, Annotated Draft Version 4.0 IR-MA-1364-3, 1993

[13] Takeuchi, A.: Object Oriented Description Environment for Computer Hardware. IFIP, 1981

[14] Dunlop, D. D.: Object-Oriented Extensions to VHDL. Proceedings of the VHDL International User‘s Forum, 1994

[15] Willis, j. C.; Bailey, S. A.; Newshutz, R.: A Proposal for Minimally Extending VHDL to Achieve Data Encapsulation Late Binding and Multiple Inheritance. Proceedings of the VHDL International User‘s Forum, 1994

[16] Covnot, B. M.; Hurst, D. W.; Swamy, S.: OO-VHDL An Object Oriented VHDL. Proceedings of the VHDL International User‘s Forum, 1994

[17] Kumar, S.; Aylor, J. H.; Johnson B. W.; Wulf, Wm. a.: Object-Oriented Techniques in Hardware Design. Computer, June 1994, p. 64-70, 1994

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

[19] Posch, K.C.: CHDL++: Understanding VHDL implementation concepts by using C++. Fifth EuroChip Workshop on VLSI Design Training. Dresden, 1994

[20] Bergé, J-M.; Fonkoua, A.; Maginot S.; Rouillard, J.: VHDL‘92 The New Features of the VHDL Hardware Description Language. 1993

[21] Gajski, D., D.; Vahid, F. Narayan, S.; Gong, J.: Specification and Design of Embedded Systems, 1994