A Flexible Message Passing Mechanism for Objective VHDL[1]

Wolfram Putzke-Röming, Martin Radetzki, Wolfgang Nebel
OFFIS, Germany
putzke@offis.uni-oldenburg.de

Abstract

When defining an object-oriented extension to VHDL, the necessary message passing is one of the most complex issues and has a large impact on the whole language. This paper identifies the requirements for message passing suited to model hardware and classifies different approaches. To allow abstract communication and reuse of protocols on system level, a new, flexible message passing mechanism proposed for Objective VHDL[1] will be introduced.

1 Structure of the paper

The introduction explains the general purpose and issues of message passing. Additionally, some basic terms are introduced. The third chapter illuminates different aspects of message passing to identify the related requirements—especially for hardware design—and shows the implications to the whole language.

To allow an estimation of message passing mechanisms, a classification scheme is proposed in Chapter 4. Chapter 5 describes and classifies two message passing mechanisms of currently most discussed approaches for object-oriented extensions to VHDL. Finally, a very flexible message passing mechanism for a new object-oriented extension of VHDL—Objective VHDL—is proposed and classified.

2 Introduction

A (hardware) system can be described in the object-oriented fashion as a set of interacting or communicating (concurrent) objects. In consequence to encapsulation the objects tend to be relatively autonomous and only loosely coupled with their environment [13]. Further, an object should contain most of the elements it needs to perform its functionality. This property provides high potential to reuse objects in other environments than the original one. On the other hand the specialization and structural decomposition of objects requires communication among the objects. The communication enables objects to use services of other objects, to inform other objects about something, or in concurrent domain to synchronize with each other. Such a communication mechanism is called message passing in the object-oriented domain. The basic idea is that objects are able to send and receive messages to provide or get some information. Of course, it is desirable that a message passing mechanism preserves as much as possible of the loose coupling of the object with its environment in order to obtain universally reusable objects.

Generally, for the communication of concurrent behaviours/processes two mechanisms can be used [6]. The first is communication using shared memory and the second the message passing via channels or communication pathways. For the communication of concurrent objects the latter choice seems to be more appropriate because passing messages is one of the basic concepts of the object-oriented paradigm.

Figure 1 shows an abstract picture of message passing. Object X in the role of a client needs a service of object Y in the role of a server. To invoke the corresponding method of Y, X sends a message via a communication pathway to Y. The message exchange is controlled by a protocol. After Y has received the message, the correct method has to be invoked. This functionality is provided by a dispatcher. As indicated by the figure we will consider the dispatching as part of the message passing mechanism. After execution of the method return values—if any—are replied.

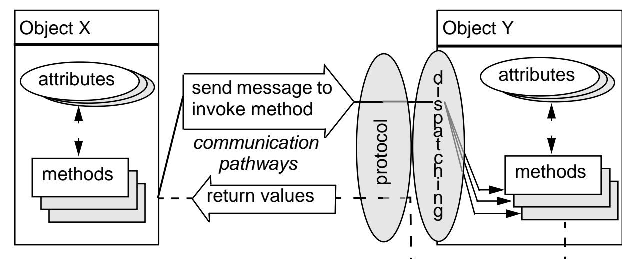

Figure 1: Message passing

1. This work has been funded as part of the OMI-ESPRIT Project REQUEST under contract 20616

In the sequential domain often the terms message passing and method call are used as synonyms. But in concurrent domain both terms have different semantics. While with the method call the invoked method needs the computational thread of the caller to execute the method, invocation of a method via message passing does not need the computational thread of the caller because the target object can use its own.

3 Aspects of message passing

In this chapter several aspects of message passing will be considered separately. This consideration shows on the one hand the design space for message passing and on the other hand the large impact to the whole language.

3.1 Consistency to language

Message passing is an integral part of an object-oriented language. The relationship between message passing and the other parts of the language is bilateral. While it is indispensable that message passing is consistent to the other object-oriented concepts, these concepts can be used for the specification of message passing. If for example an language embodies different encapsulation concepts (classes, objects) message passing has to take this in account—at least its implementation. A class concept based on abstract data types needs another realization of the message passing compared to a class concepts which represents (structural) hardware-components. The message passing mechanism proposed for Objective VHDL in this paper will give an example how the object-oriented concepts like classification, inheritance, and polymorphism can be used to implement it.

Another desired feature of message passing is the consitency with the techniques of object-oriented modelling. This means consistency in terms of refinement and extensibility. If a model is extended by components which need a more refined or different way for communication, the message passing mechanism must be adaptable and by object-oriented means.

Further, consistency means that the abstraction level of message passing fits to the abstraction levels the whole language is intended for. The programming interface should be abstract and easy to use. For example, the appropriate encapsulation of the transmission of messages by send/receive methods.

Abstraction and encapsulation, however, shouldn't be considered in isolation because they have large impact on other aspects of message passing (e.g. flexibility, simulation/synthesis).

3.2 Communication pathways

Passing a message from one object to another and performing the communication protocol requires a communication pathway which interconnects the objects. If communication is restricted to 1:1 relation[2] the target object can be identified directly by the communication pathway. But the abstraction and definition of communication pathways differs significantly in literature.

In case a method call has the semantics of a procedure/function call, the procedure/function call mechanism and the name of the target object/method can abstractly be considered as the communication pathway.

If the communicating objects represent hardware components, in VHDL terms—entities, another representation of communication pathways is necessary. In [14] some kind of identifier for an entity object, called a handle which can be exchanged among entities, is proposed. If an object has the handle of another object, the handle can be used to address the other object to pass a message. So the handle can be considered as the communication pathway. This solution is abstract because this communication pathways have no direct physical representation and flexible because it allows to establish communication pathways during runtime. Generally, it allows to send messages to objects which are dynamically generated during runtime. The dynamic generation of objects, however, might be a powerful feature for system design, but it is really far away from hardware.

Another solution proposed in [12][9][11] is to use the VHDL mechanism to exchange data between components, i.e. to interconnect components by signals. From the sender's point of view the target object can be addressed by the port which connects both objects. Of course, this approach isn't as flexible as the handle solution to address component-objects, but it is very close to hardware.

Although signals can be used for communication, there is still a gap between the abstraction provided by VHDL signals and communication in object-oriented sense.

3.3 Protocol

In software, sending a message to an object has the semantics of a procedure or function call[3]. Results can be given back by assignments [12]. In hardware message passing among concurrent components/objects needs specialized protocols. Of course, the necessity of protocols results in a much tighter relation between the objects than it is desired by object-oriented paradigm (cf. Chapter 2). Sending a message needs the knowledge and the ability to perform the target object's communication protocol.

2. A 1:1 relation not necessarily means a point to point communication because an object can consist of concurrent processes.
3. Without consideration of distributed programs.

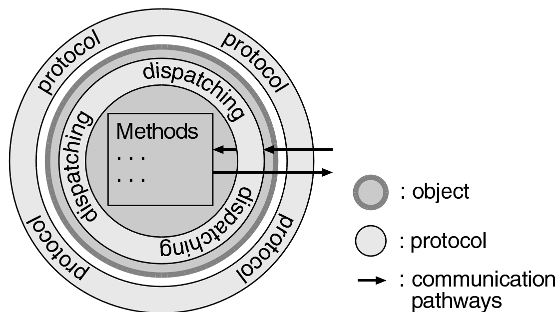

Figure 2: Encapsulation of an object by protocol

From a communication point of view the protocol can be seen as the encapsulation of an object (Figure 2). In the following subchapters several aspects of a protocol for message passing will be illuminated

3.3.1 Abstraction

Since the object-oriented paradigm addresses modelling on higher abstraction levels, the details of the communication protocol should be encapsulated and the specialization of the protocol has to correspond with the abstraction. The protocol should be applied through an abstract interface.

3.3.2 Flexibility

A universal message passing mechanism for hardware design needs the possibility to integrate different protocols for message exchange. An abstract model of an MMU on system level may need another protocol than a simple register on RT level. Further, in perspective of a top down design methodology, it should be possible to refine a protocol towards more detail according to the description level. The same need occurs if co-simulation of abstract models together with already synthesized models is desired.

The flexibility, however, to choose or refine a new protocol is at the expense of encapsulation of the protocol. Even if the interface to use a protocol can be encapsulated it is not possible to hide the protocol by the language completely.

3.3.3 Synchronization

In concurrent object-oriented domain objects need to synchronize with each other to describe their behaviour dependent on the state of other objects.

We would like to differ three synchronization modes:

- synchronous
- asynchronous
- data-driven

With synchronous communication the sender object needs to wait until a receiver object is ready to receive a message. In most synchronous communication mechanisms [7] the sender object/process is blocked from the moment of sending a message request until the service which is intended to be invoked by the message is finished and the results are given back.

With asynchronous communication the sender does not wait for the readiness of the server object to receive a message. In order to avoid the loss of messages, this mode requires queuing of messages within the communication pathway or the receiver. An additional advantage of such queues is the potential flexibility to dequeue messages in another order than FIFO. On the other hand the message queues can have large impact on simulation and synthesis aspects. Generally, asynchronous communication is non-blocking [3]. So the sender object can continue its computations directly after sending. The return of results requires to send explicitly a message from the server to the client.

The data-driven synchronization [5] allows a sender object to run its computations until the results of a previously sent message request are needed. In this case the sender has to wait until these results will be provided by the server object. In literature the initial sending of a message request is described to be asynchronously [3]. However, a synchronous (but non-blocking!) sending of a message request is also conceivable.

In summary, the data-driven synchronization allows more flexibility than standard synchronous/asynchronous communication.

3.3.4 Concurrency

To be consistent with (Objective) VHDL, a message passing mechanism must preserve and support the concurrency provided by (Objective) VHDL.

The relation between communication and concurrency is ambivalent. On the one hand concurrent objects/processes are the reason for the necessity of message passing, on the other hand the message passing mechanism can restrict the concurrency. If an object contains only one process (dispatcher process) which receives requests and dispatches them, all requests will be sequentialized.

For concurrent object-oriented languages it is expected that the objects can have own activity and can run in parallel. But besides the concurrency of parallel running objects there can be concurrency inside the objects if they contain concurrent processes. This intra-object concurrency may allow an object to execute requests for services in parallel.

For example, a dual-ported RAM allows concurrent read and write operations. This can be modelled with concurrent dispatching processes. However, allowing parallel method execution raises the potential problem of nondeterministic behaviour of the object, due to concurrent access to the same attributes. But VHDL already provides mechanisms to solve concurrent access to signals and variables.

Concurrent (write) access to signals can be handled by resolution functions. With variables the proposed protected types can be used [15]. So at least atomic access to variables can be ensured. But the problem with nondeterminism is still unsolved because the value of a shared variable depends on the activation order of the accessing processes.

A possibility to avoid nondeterminsim is to ensure that concurrent methods have only access to exclusive attributes. This can be implemented by grouping all methods which have access to the same attributes. Each group gets an own dispatching process and maybe a queue. So only methods without access conflicts can be executed in parallel.

3.4 Simulation/Synthesis efficiency

Simulation efficiency and synthesizability are general aspects for the quality of object-oriented extensions of a HDL. These are being influenced by implementation decisions of message passing/protocol and communication frequency between objects. A complex protocol enriched with a lot of detailed timing informations and unlimited message buffers to avoid the loss of messages may decrease the simulation speed significantly. Moreover, without limitation of maximal buffer size the protocol is not translatable into hardware.

4 Classification scheme

In the above chapters some of the aspects a message passing mechanism for an object-oriented HDL has to deal with have been proposed and discussed. These can be used to define a classification scheme for different message passing mechanisms.

The first criterion for classification is the flexibility of the message passing mechanism. The following cases will be distinguished:

- *flexible*: different protocols are possible and can be refined.
- *fixed*: the protocol is not modifiable.
- *semi-flexible*: one protocol, with potential for refinement or different not-refinable protocols.

The second criterion is the ability of an object to accept and perform several requests concurrently. It should be only distinguished between:

- *yes*: it is possible.
- *no*: it is not possible.

The third criterion is the synchronization of the message passing. It can be:

- *blocking*: the sender is always blocked until the return values are received.
- *non-blocking*: the sender continues execution after sending a request. Maybe at a certain point of execution he has to wait for the results.
- *both*: depending on the kind of message or on the kind of method invocation (method call vs. message passing, cf. Introduction) blocking or non blocking communication is possible.

The last criterion is whether there are queues to buffer messages if the receiver is busy. It will be distinguished between:

- *no*: no queues are provided.
- *one*: one queue per object.
- *many*: more than one queues per object.

flexibility	parallel methods (per object)	synchronization	queues (per object)
flexible fixed semi-flexible	yes no	blocking non-blocking both	no one many

Table 1: Classification scheme

5 Other OO-VHDL approaches

During the last years several proposals for an object-oriented extension to VHDL have been made [1][2][4][12][14]. All the proposals need to define message passing to be object-oriented.

Because not all of the proposals can be considered here, two typical but completely different proposals are selected to describe their message passing mechanisms. Because this cannot be done isolated, it is necessary to introduce the core concepts of the proposals up to a certain degree. But it is outside the scope of the paper to draw a complete picture of the special proposals.

The Vista approach [14] introduces a new design unit called Entity Object (EO) which is based on the VHDL entity and it's architecture. In addition to the entity the EO may contain method specifications called operation specifications. Operations are similar to procedures, but they are visible outside the EO. In contrast to procedures operations can have a priority and a specified minimal execution time. For invocation of operations the EOs are not interconnected with explicit communication pathways (signals).

To address an EO, each EO has an accompanying handle which is a new predefined type that can be stored in signals or variables and can be part of composite types. Handles can be exchanged to make the corresponding EO addressable for other objects. A special handle to address an object itself (self) is predefined. The parent class can be addressed by the new keyword ‘super’. A message send request is performed by a send statement which includes the handle of the target EO, the name of the operation, and the parameter values.

Each EO has one queue to buffer incoming messages. The messages in the queue are dequeued by their priority. Messages with the same priority are dequeued by FIFO. One queue per EO means that concurrent requests are se-

quentialized. If an EO needs to invoke an own operation (send self), this request will not be queued. It will be treated like a procedure call and immediately executed. This mechanism avoids deadlocks with recursive method calls.

flexibility	parallel methods (per object)	synchronization	queues (per object)
fixed	no	both	one

Table 2: Classification of [14]

Messages can be of blocking (default) or non-blocking mode (immediate). But immediate messages are restricted to have in-parameters only. The blocking mode cannot be changed during inheritance.

For EO synchronization a rendezvous concept is provided. Accept and select statements similar to Ada are used to establish a rendezvous.

Finally, it should be remarked that the proposed message passing mechanism is neither sythesizable nor it is intended to be. The abstract concepts of dynamic communication pathways represented by the handle concept and the unlimited message buffers have no counterpart in hardware.

Another approach was developed at Oldenburg University [12]. It is based on the VHDL type concept. Records are used to represent the objects. To allow a record to be expandable by inheritance, it is marked as a tagged record. The corresponding methods, which are simple procedures, must be defined in the same design unit (package). Because the methods cannot be formally encapsulated by the object it belongs to, the parameter list of each method contains a parameter of the object's type which assigns the method to the object. For each tagged record a corresponding class-wide type exists, which is the union of all types derived from the tagged record. With the attribute 'CLASS the class-wide type can be referenced. Polymorphism is based on the class-wide types. If a method is called with an actual of class-wide type (mode in, inout), the actual type is determined during runtime and the correct method will be invoked.

Inter-process communication can be modelled consistently with VHDL by signals. Abstraction and expandability are supported by use of polymorphic signals (class-wide type). Sending a message to another object has the semantics of a procedure/function call or in our terminology a method call. The requested method is executed by the sender which is blocked until the end of the request. Several methods of an object can be performed concurrently if they are requested by different processes. But in case of concurrent assignment to a signal (instantiation of a tagged record), resolution functions are necessary.

Even if in [12] a special protocol mechanism is proposed other protocols can be integrated because the protocol is not built into the language. The classification here refers to the proposed master-slave protocol.

flexibility	parallel methods (per object)	synchronization	queues (per object)
flexible	(yes)	blocking	no

Table 3: Classification of [12]

6 Objective VHDL

Objective VHDL is the object-oriented extension to VHDL developed in the EC-Project REQUEST[4] [8][10].

Objective VHDL combines the structural object approach [14] with the type object approach [12]. Both language extensions have shown their suitability for hardware design.

The structural objects are usual VHDL entities. Attributes and methods of an entity class are declared within the declarative part of the entity. They correspond to VHDL object declarations (signals, shared-variables and constants) and procedure or function declarations respectively. The implementation of the methods follows in the corresponding architecture. Single inheritance for entities and architectures is provided but no polymorphism on entity objects.

A type class consists of a declaration and a definition, likewise. Class types are declared like usual types but the declaration of attributes and methods are assembled between the new 'is class' and 'end class' constructs. The implementation of the declared methods or private methods, which are not declared in the interface, follows in the corresponding class body. As well as for the entity classes, single inheritance is provided for the type classes.

Each class type has an associated class-wide type, marked by a new attribute 'CLASS. The class-wide type is the union of the type itself and all derived subclasses. Similar to the [12] approach class-wide types are used to realize polymorphism on type classes. A variable or signal of class-wide type T'CLASS can hold any instantiation of a class which is derived from T or T itself.

Calling a method of a directly visible instantiation of a class type has the semantics of a simple procedure/function call (blocking). Calling a method of an entity object or a type object instantiated in another entity is more difficult. A method of an entity cannot be called directly because the entity encapsulates the procedures/methods completely. The only interface to the outer world are the ports and generics of the entity. Breaking this encapsulation would

4. Objective VHDL is defined in [9]. The Objective VHDL Language Reference Manual is intended as an extension to the VHDL LRM. Although the current status of Objective VHDL within the REQUEST project is stable minor changes in the language are possible in future.

change VHDL. A solution to that problem, which was chosen in [14], was to introduce a new design unit—the Entity Object where the methods (operations) of an EO are visible outside the EO. Due to the additional implementation costs for a new design unit, this solution was discarded for Objective VHDL.

6.1 Message passing mechanism

To provide the flexibility to use an appropriate communication for message exchange, the message passing is not fixed in the language definition. Nevertheless, a way to implement a flexible message passing will be shown by usage of the other object-oriented features. The main ideas will be now described in more detail.

Basically, the message passing mechanism consists of three parts:

- the communication structure which defines the connection of objects with communication pathways (Figure 3),
- protocol and messages for exchange (Figure 4),
- dispatcher (Figure 3).

6.1.1 Communication structure

To enable communication between objects which are represented by entities or type objects inside different processes, the communication pathways between the objects are implemented by VHDL signals. Beside the signal which carries the message additional signals for the protocol may be necessary. To avoid resolution functions, the signals are unidirectional. Consequently, this requires two opposite communication pathways if the message request produces return values. This physical connection is depicted in Figure 3.

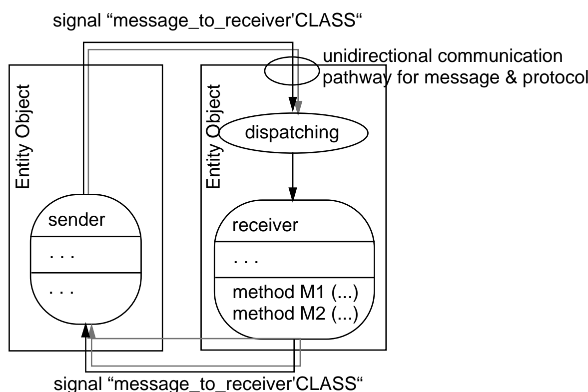

Figure 3: Communication structure

So the target object of a communication can be addressed by the port which connects the sender with the receiver. Further a communication pathway is restricted to connect only two objects.

However, VHDL signals do not provide the abstraction expected from object-oriented message passing. To enable the connecting signals to provide the required abstraction, they will be implemented as polymorphic type classes. So such a signal gets the ability to hold different messages and to encapsulate a communication protocol.

The messages and the communication protocol can be modelled by a structure which is shown in Figure 4. But in following main emphasis is given to the modelling of the messages.

6.1.2 The messages and the protocol

The messages and the protocol are implemented and encapsulated by an abstract type class (‘message’). The class is abstract because it is not intended to be instantiated and it should serve as base class for the classes representing real messages. Furthermore, the class provides the interface which allows to apply the message passing among the objects. The interface is inherited to the derived classes and declares methods like ‘send’ (a message), ‘receive’ (a message, or ‘dispatch’ etc. .

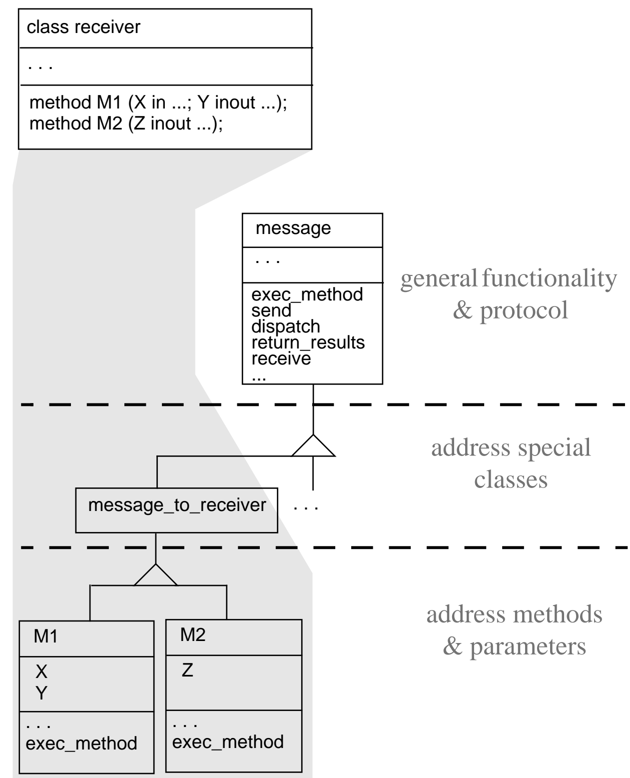

Figure 4: Modelling of messages

To enable a communication pathway to hold all messages to a target object, the type class ‘message’ is refined by inheritance. Because (target) classes differ in the methods their instantiations can receive, for each kind of (target)

object an abstract subclass is derived (Figure 4, 2nd-stage). This class implements no additional functionality, its purpose is only to distinguish the different kinds of objects. (Different objects which are instantiations of the same class are represented only one time!)

Finally, a message must correspond to one method which should be invoked and the message must contain the parameters for method invocation. To allow this, each class of the second stage of Figure 4 is refined with subclasses representing the methods of the (target) object (Figure 4, 3rd stage). The subclasses are named by the methods and the method parameters are represented as attributes.

A signal which is an instantiation of such a class can hold a message corresponding to a special method and provides the communication protocol if one was defined in the superclasses.

A communication pathway, however, should be able to hold all messages which can be sent to an object. By instantiation of the communication pathway as signal of a classwide type belonging to the representation of a kind of object (Figure 4, 2nd stage) this ability is given to the signal.

Finally, each of the classes representing a method (Figure 4, third stage) implements a method ('exec_method'), which invokes the corresponding method of the target object. As mentioned before, the methods of an entity object cannot be invoked directly because the entity encapsulates its methods completely. Nevertheless, to allow this method invocation a new construct 'for entity ... end for' is introduced. The semantics of the construct is to make the declarations of an entity class visible inside a type class in order to allow its invocation. The implementation of such a method must be available wherever 'exec_method' is invoked with an instance of the message class.

Results produced by the method execution can be sent back by the same mechanism.

6.1.3 Dispatching

Each entity object contains one or more dispatching processes. These processes are sensitive to the ports which carry the incoming messages and must be specified by the user. To dispatch the incoming messages, functionality provided by the class 'message' can be used if implemented (method 'dispatch'). But basically, within the dispatcher process only the method 'exec_method' of the received message has to be invoked which calls the desired method of the entity object. By the type of the received message it can be decided during runtime which 'exec_method' has to be called.

The number of the dispatching processes can be chosen by the user and determines the number of concurrently executable methods. If concurrent methods are allowed, the user has to take care about conflicting concurrent access to the attributes. To ensure atomic access to attributes, shared variables with the protect mechanism (cf. Chapter 3.3.4)[15] can be used.

6.1.4 Synchronization

Synchronization can be very flexible. Synchronous, asynchronous, and data driven sychronization (c.f Chapter 3.3.3) can be modelled.

The proposed mechanism does not require that an object is blocked after sending a message to a concurrent object. But it should wait at least until it is ensured by the protocol that the target object accepts the request. Of course, additional synchronization is necessary if the results of the request are needed.

For asynchon communication message queues have to be integrated into the message passing mechanism.

6.2 Summary of message passing

The provided message passing mechanism provides easy means to send messages. The sender identifies the target object through the port(s) by which they are connected. Now, it needs only to call the 'send' method of the class 'message' with the port names and the message itself as parameters. The 'send' method will perform the protocol and assign the message to the interconnecting signal if the receiver is ready to accept it. After sending the sender can be sure the message is received and can continue its execution until potential results of the request are needed. In this case the sender is blocked until the results are available.

The receiving object possesses a dispatching process which is sensitive to the connecting port signal. If a new message arrives, the dispatching process performs the receiver's part of the communication protocol and invokes the desired method. Of course, this functionality can be encapsulated by a 'receive' or 'dispatch' method of the class 'message'.

The potential results can be send back implicit and immediately after their computation or explicit by a new communication.

Finally, it is important to note that the modelling of messages (cf. chapter 6.1.2) is really straight forward. So the effort to use this communication mechanism can be reduced significantly by a tool which produces the messages automatically.

6.3 Classification

According to the proposed classification scheme the message passing can be classified as shown in Table 4. Different protocols can be defined and stored in a library. If required they can be refined. Concurrency of method execution depends only on the number of dispatching processes the user defines. In the proposed communication mechanism a method call is blocking and passing a mes-

sage can be blocking as well as not blocking. Queues to buffer messages and to allow asynchronous communication can be modelled in principle but are not integrated in the communication up to now.

So it is important to note that a special implementation of the message passing mechanism may result in a slightly other classification.

flexibility	parallel methods (per object)	synchronization	queues (per object)
flexible	yes	both	(no)

Table 4: Classification of Objective VHDL

7 Future work

A precompiler which translates Objective VHDL to VHDL will be implemented. With the availability of this tool the desired benefits of Objective VHDL and especially the message passing mechanism can be evaluated.

The proposed mechanism for message passing has potential for improvements. For better protocol reuse a stronger separation of the protocol and the messages will be useful. The modelling effort which is caused by the restriction of unidirectional communication pathways can be reduced if resolution functions for the connecting signals can be defined.

8 Conclusion

By analyzing and discussing the several aspects of abstract communication the design space for message passing has been shown and a classification scheme for message passing mechanisms has been developed. The scheme has been applied to the message passing mechanisms of two of the currently most discussed proposals for object-oriented extensions to VHDL.

Finally, a new idea for message passing mechanism developed for Objective VHDL was introduced and classified. The new approach targets especially flexibility of protocols, reuse of protocols and consistency to VHDL (concurrency).

9 References

[1] Peter J. Ashenden, Philip A. Wilsey, Dale E. Martin: *SUAVE: Painless Extensions for an Object-Oriented VHDL*. VUIF Fall ’97 Conference Proceedings

[2] Judith Benzakki, Bachir Djafri: *Object Oriented Extensions to VHDL - The LaMI proposal*, submitted to CHDL’97

[3] L. Bergmans, M. Aksit, K. Wakita, A. Yonezawa: *An Object-Oriented Model for Extensible Concurrent Systems: The Composition-Filters Approach*, http://wwwtrese.cs.utwente.nl/Docs/Tresepapers/tresepapers.html

[4] David Cabanis, Prof Sa'ad Medhat: *Object-Oriented Extensions to VHDL: The Classification Orientation*, VHDL User Forum, SIG-VHDL Spring’96, Dresden, Germany

[5] D. Caromel: *Concurrency And Reusability: From Sequential To Parallel*, JOOP, September/October 1990

[6] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, Jie Gong: *Specification and Design of Embedded Systems*, Prentice-Hall, Inc., 1994

[7] C. A. R. Hoare: Communicating Sequential Processes, Prentice Hall Int. Series in Computer Science, 1985

[8] W. Nebel, W. Putzke-Röming, M. Radetzki: *Das OMI-Projekt REQUEST*, Eingeladener Vortrag, 3. GI/ITG/GME-Workshop, Hardwarebeschreibungssprachen und Modellierungsparadigmen, Holzau, 26.-28.02.1997.

[9] Serge Maginot, Wolfgang Nebel, Wolfram Putzke-Röming, Martin Radetzki: *Final Objective VHDL Language Definition*, REQUEST deliberable 2.1A, public May 1997, available on WWW from URL http://eis.informatik.uni-oldenburg.de/research/request.html

[10] M. Radetzki, W. Putzke-Röming, W. Nebel, J. M. Bergé, A. M. Tagant, S. Maginot: *VHDL extensions to support abstraction and reuse*, 2nd Workshop on libraries, Component Modeling, and Quality assurance, Toledo, April 1997

[11] Martin Radetzki, Wolfram Putzke, Wolfgang Nebel: *Language Architecture Document on Objective VHDL*, REQUEST deliverable 1.2C, public, December 1996

[12] Guido Schumacher, Wolfgang Nebel: *Inheritance Concept for signals in Object Oriented Extensions to VHDL, Euro*-DAC’95 with Euro-VHDL’95, Brighton, England (1995)

[13] Bran Selic, Garth Gullekson, Paul T. Ward: *Real-Time Object-Oriented Modeling*, Wiley 1994

[14] Sowmitri Swamy, Arthur Molin, Burt Convot: *OO-VHDL Object-Oriented Extensions to VHDL*, IEEE Computer, October 1995J. Willis, S. Bailey, C. Swart: *Shared Variable Language Change Specification* (PAR 1076), Version 5.7, December 1996

[15] J. Willis, S. Bailey, C. Swart: *Shared Variable Language Change Specification* (PAR 1076), Version 5.7, December 1996