Object-Oriented Hardware Modelling – Where to apply and what are the objects?[1]

Wolfgang Nebel
OFFIS
Escherweg 2
D-26121 Oldenburg, Germany
nebel@offis.uni-oldenburg.de

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

Abstract

The importance of reusability of hardware models for the necessary increase in design productivity will be explained for different modelling problems. Methods having previously been proven to be successful in software engineering will be analysed with respect to their applicability to hardware design. It will be shown that object-oriented modelling techniques do potentially increase design productivity, but that VHDL in its current version does not support object oriented modelling. Possible subjects to object-orientation will be discussed.

1. Introduction

The -despite of earlier assumptions- not saturating integration density of integrated circuits provides sufficient motivation to investigate further possibilities to increase design productivity. The increase of complexity can be estimated to about ten every seven years. It can not be compensated by the increase of computing power, which on the one hand without a change in design methodology does not achieve an increase in intellectual value addition, on the other hand is consumed mostly by complex datamangement systems and user friendly graphical interfaces.

As keys to manage the design complexity and the increase of design effort the terms *Hierarchy* and *Abstraction* are typically mentioned. A hierarchical, structural decomposition of the design problem reduces the complexity by isolating subproblems and hence makes the global design problem accessible for a solution. This achievement has to be paid with a suboptimal total solution which is due to the local optimization of the substructures.

Abstraction should be looked at from two points of view. First it allows to encapsulate existing components which can be described by an abstract model containing only that information which is required at the higher level of abstraction. Examples of this kind of abstraction are cell libraries of ASIC vendors. Here abstraction allows the reuse of hardware components. They don't need to be redesigned for each application, i.e. the design cost -measured in terms of transistors- is reduced.

On the other hand abstraction can be used for reducing design effort if design detail can automatically be attributed to less detailed design specifications using synthesis tools. Application examples are: logic synthesis, technology mapping, place & route. Here the reduction in design effort is due to a re-use of automated design strategies and architectures.

1.1. Traditional approaches of re-use

In the past the method of design data encapsulation has been developed in an evolutionary way in the direction of higher levels of abstraction. The steps *low level design*, *encapsulation* and *re-use* were first applied to transistors, then to cells and finally to macros. On the macro level, however, the limits became obvious. To design a comprehensive library of complex parameterized macro cells has turned out to be not cost efficient for general purpose applications. The reasons are the high initial and maintenance cost of the library whose elements are possibly only seldom used and on top of that not optimal for the particular application. Today in general purpose ASIC libraries there are usually only memories and analogue cells as macro cells. Generators which were typically provided for arithmetic functions etc. have been replaced by synthesizable models a couple of years ago. Some exceptions can be found in the domain of signal processing and other special purpose libraries.

One can conclude that the necessary further increase in level of abstraction cannot be achieved by aggregating physical representations of larger functional units and abstracting them. The more promising approach is to reduce the effort of high level modelling and to build on automatic synthesis.

1. Part of this work has been funded by the ESPRIT OMI project 20616, REQUEST - REuse and QUality ESTimation: Advanced VHDL based design methodology for quick system development.

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00

The acceptance of synthesis tools depends not only on their stability, run time and achievable design quality, but also on the predictability and comprehensiveness of the results. A chaotic behaviour of the tools -i.e. minimal changes of the tool input result in major changes of the tool output- can only be avoided by a good controllability of the synthesis result. Behavioural synthesis maintains the process structure and hence provides a certain degree of controllability by limiting the scope of changes to single processes.

Moving the creative part of the design process to those levels which are covered by hardware description languages, of course also moves the largest potential of design productivity increase to those levels. According to [1] already in 1993 almost 60% of the US designer community used logic synthesis and more than 50% used hardware description languages. The large number of simulation and synthesis iterations [2] emphasizes the need to improve the modelling productivity. The current effort in this domain is due to necessary debug phases and the need to steer the synthesis process into the intended direction. Different aspects of modelling cost and productivity will be discussed in section two.

1.2. Software-engineering

Software-engineering is frequently cited to be comparable to the problem of complexity in hardware design. Abstraction and hierarchy have been applied to manage design complexity in software engineering. The latest development in rising the level of abstraction are object-oriented programming languages. In parallel to the language design object-oriented analysis and design methodologies have been developed. The object-oriented way of structuring a system at the first glance seems to be closer to how hardware designers tend to work, i.e. a structure of communicating hardware entities. However, as we will see later, the differences between these classical hardware entities and the objects of the software world are still significant.

Section three will introduce into techniques which are typically called object-oriented. The acceptance problem typically coming together with a shift in paradigms and the current state of standardization of object-oriented extensions to VHDL will be explained in section four.

2. Modelling problems

A careful analysis of the problems is required until a technique, which has proven to show good results in one domain, is applied to another domain. This principle has to be obeyed for object-oriented methods before recommending them as panacea.

Considering the special objective of reusability, modelling cost arise during different phases of the design. Their relevance and whether reusability is feasible certainly also depends on the complexity of the models. It is further necessary to identify those aspects of the design which shall be subject to re-use.

Table 1 structures these questions w.r.t. the modelling phases and the level of abstraction of the respective models. The modelling phases and their particular problems will be explained in the following.

2.1. Initial modelling cost

In this taxonomy initial modelling cost covers all cost arising in the specification phase of a particular model until it is accepted without syntactical and obvious semantic and functional errors by an analyser (simulator). The task starts with the interpretation and analysis of a specification, which may be informal, covers structuring and coding and ends with the fixing of syntactical, semantic and functional errors. The result is a baseline model of further design phases.

The initial modelling cost depends not just on the target to be modelled and the user friendliness and equipment of the work-place, but also on how well the modelling language fits the application, the objects to be modelled and the level of abstraction. The better the language fits the problem, the less code needs to be written; the modelling productivity increases. Another factor is the organization of and extent to which previously acquired design know-how can be exploited.

2.2. Cost of maintenance

Typically after the initial modelling phase some disturbances occur requiring changes of the initial model. These include e.g. changes of the specification, functional errors detected late or changes of constraints. These changes require to get acquainted with the model and to implement the modifications in such a way that exclusively the intended changes are made but no others. In particular it is necessary to minimize the impact of the amendments to other members of the design team. The effort depends on the structure and documentation of the initial model, but also on the structuring capabilities of the hardware description language used. An extensive encapsulation of data and functionality and the avoidance of having to consider the impact of local changes in non-local entities helps to increase productivity.

2.3. Cost of re-use

Re-use of models is certainly an important aspect to increase modelling productivity. Re-use is in particular in the following situations advantageous [4]: frequently used modules, e.g. memories; standard functionality, e.g. protocols; evolutionary design (about 80% of all designs are re-designs). A re-use concept for such components has been presented in [3]. The components are supplemented by the support of the phases of the design concept. As correctly stated in [4], a pure re-use of an existing module does not achieve any added value. I.e. for innovations, the modules

	initial spec.	system	subsystem	cell level
initial modelling cost	moderate	high	moderate	low
maintenance cost	moderate	high	high	low
cost of re-use	moderate	high	high	low
cost of model disposal	n.a.	n.a.	high	high
level of re-use	very low	moderate	high	very high

Table 1: Importance of modelling cost and re-use[1]

have to be adaptable to e.g. new technologies or extended functionality. Technology migration can be achieved by synthesizable models. A certain flexibility in the application can be gained by parameterized models, however, it is limited if the basic functionality of the model needs to be extended or modified.

An analysis of the cost of re-use is required in order to decide whether the re-use of a model for a certain application is feasible. The cost is composed of the additional cost of a reusable model compared to a non-reusable model. This includes the effort for better quality assurance measures and possibly increased cost of more flexible modelling. Secondly, cost of re-use includes the cost of actually reusing a model. This covers the search for the required model, the necessary modifications for the actual purpose and the possibly increased hardware cost which is due to overhead compared to an optimized full custom model.

2.4. Cost of model disposal

At first glance it may sound strange to think about a disposal problem in the context of models. Cost of disposal here is cost of pruning the repertoire of maintained models inside an organisation. Such situations can arise if e.g. software licences or hardware maintenance contracts cannot be cancelled because they are needed to maintain, update etc. design data including models. It is realistic to assume that many systems need to be supported although they are technically obsolete just because they are required to fulfil long term delivery contracts of certain ICs.

Disposing these models requires an inventory where these models are used and a replacement of the model instances by functionally equivalent alternatives. As in the other cost domains getting acquainted with the models and their modification is part of the disposal cost.

2.5. Levels of complexity

The level of complexity of a model certainly has a major impact on the modelling cost and the feasibility of reusing such a model. Rather than defining the complexity of the different levels in absolute terms, e.g. number of equivalent logic gates, it is more appropriate to define a generic taxonomy of complexity which is independent of the current integration capabilities of microelectronics technology. While the number of gates which today build a complete system will be needed for a subsystem in few years, the generic characteristic of a system to be composed of subsystems is time invariant.

2.6. Initial specification

Each design process with the exception of minor redesigns will start with the capture of the interfaces, constraints and global functionality of the system. In many cases this specification is not yet based on a hardware description language. This model is the interface between marketing/customer and designer.

2.7. System modelling

The next step of the design is the formal specification of the system behaviour in terms of a hardware description language. The result is a simulatable, but in general not yet synthesizable specification. Different specification mechanisms are often used to create the model or parts of the system, e.g. state transition diagrams, time diagrams etc. The model consists of one or more behavioural level processes. It is subject to be validated by simulation and builds the baseline of the next design step, the structural decomposition into subsystems and their communication.

2.8. Subsystem modelling

A subsystem comprises a major functional unit of a system. Again different methods may be used to generate the models. In contrast to system models, however, the probability of reusing subsystem models is much larger. Different combinations of cores and periphery subsystems may be useful for different applications and a value addition may be created simply by new combinations of subsystems into systems. In order to enhance the reusability, functional modifications should be possible at low cost and the functionality should be completely encapsulated to support to get easily acquainted with the model. A methodology for behavioural component re-use based on encapsulation of the functionality has been presented in [5]. In many cases subsystems will be modelled for re-use from the beginning. Even if not achievable with current synthesis technology, subsystem models should be targeted to synthesizability.

1. This first estimation will be refined in course of the ESPRIT OMI project REQUEST.

2.9. Cell modelling

The basic building blocks targeted during logic synthesis and technology mapping are standard cell models. The functional modelling of these cells is straight forward. The maintenance is limited to the (semiautomatic) characterization of new processes and the adaptation to new design frameworks. Of course these cells are only designed for re-use, however, the effort to really re-use them is minimal due to their simple functionality. Furthermore the cell re-use is usually done by tools rather than by designers.

2.10. Best gain domains of re-use

From the previous ideas about the different importance of modelling cost and the relevance of re-use at the different levels of complexity it can easily be concluded, that the reductions in cost of re-use at the subsystem level are most promising. Subsystems are likely to be re-used, hence they inhibit the potential of significant total design efficiency increase. At the same time, the cost of reusing subsystems is regarded high because they are functionally complex, expensive in initial modelling, shared in design teams and subject to functional modifications and technology migrations. Additionally applying new tools and design methodologies to this level is a natural extension of the bottom up evolution of the design automation process.

From the modelling efficiency point of view, a reduction in cost and development time of the initial specification is also of high priority and their suitability for efficient re-use is very important.

3. Object-oriented techniques

The result of the analysis of modelling cost as given in section two suggests to check to what extent modelling cost can be reduced by applying object-oriented methods to hardware design.

3.1. Introduction

The fundamental concept of object-oriented programming is a system of communicating objects. Each object is characterized by attributes whose values define the state of the object. The attributes' values can be modified by a repertoire of methods. The communication amongst the objects is done by invoking respective methods of the target object. Object-oriented programming techniques are usually characterized with buzz-words: *class*, *inheritance*, *object*, *polymorphism*, *method*, *encapsulation* and *message passing*.

Objects are instances of a class. Inheritance allows to create a class hierarchy in which common features of several classes are specified in a parent class. This class hierarchy and the inheritance concept allows to specialize and generalize objects. Both support the re-use of models. Creating variants of objects in non-object-oriented modelling techniques is usually done by copy, modify and paste. This results in a multitude of versions which differ in details only, contain a lot of redundancy and are difficult to administrate. In particular problems occur when modifications are needed in parts of the models which are common to all different versions. The consistency of the model database can only be maintained at high cost and risk, because such modifications need to be made in many models and it is difficult to guarantee not to introduce side effects. A clear advantage of a class hierarchy and an inheritance concept is the better reusability of objects, through the easier administration of the version multitude.

Polymorphism is the capability of an object-oriented language to handle several versions of a method, i.e. operations in different classes of a class hierarchy, with the same name. In contrast to the VHDL type of operator - overloading the particular instance of the method to be invoked is determined at run time of the model (late binding). The criterion for this selection is the actual class of the target object. In VHDL operator-overloading the binding is done statically at elaboration time depending on the number of parameter associations, the types and order of parameters, the names of formal parameters and the return type of function calls. This early binding does neither allow to dynamically compute the target object during run time nor to have non-static parameter types.

Assuming one has to design a modular instruction set architecture which supports several classes of instructions and address modes. All instruction classes are defined in a class hierarchy as subclass of a generalised class *instruction*. The instruction counter addresses an object which is an instance of a class being derived from *instruction*. All objects of class instruction provide the method *execute*. Invoking this method of the actual instruction being addressed by the instruction counter would result in the execution of the particular instruction although all execution phases of the architecture would be started with the same statement e.g.: *execute (<instruction counter>)*. Any later additions to the instruction set do not influence the code of the control flow of the sequencer.

A similar realization in VHDL with different types of each class of instructions and address modes is not possible, because overloading requires that the types of the procedure arguments need to be static during elaboration time. Hence a single type instruction would have to be defined which can accommodate all possible instructions. Each extension of the instruction set and each new address mode would require an update of this type and -even worse- of all references to this type. It can be concluded that the more efficient extensibility of polymorphic methods and objects would be advantageous in evolutionary hardware design.

As stated earlier, an object can be viewed as a finite state machine with state attributes (instance variables) and state transition and output functions (methods). The instance variables can be accessed by sending externally visible methods of the object to the target. If this is the only possible access, the instance variable is encapsulated, since the client of the object does not need insight into the

internal structure of the object. This encapsulation not only provides an abstraction, but reduces implementation dependencies between objects to the set of visible methods.

3.2. Objects in hardware design

It is the intention of this section to elaborate what are suitable objects in hardware design for the application of object-oriented modelling methods. As stated earlier, physical objects have traditionally been the subject of the design process. This has been obvious, because each single design step has been a bottom-up design step, even in a globally top-down design flow. This statement is at least valid for the lower levels of design where a design step is usually a structural composition of predefined elements. These may be available partly in abstract form, e.g. as DesignWare, CAD-algorithm etc. Even a RT-level model is a structural model of elements of a fixed repertoire of predefined operators.

An object-orientation of a design flow which is based on physical components suggests to consider VHDL entities as objects. They would have to be provided with the respective capabilities of object-oriented programming. Inheritance could support the derivation of one entity from another with the addition of new services (methods). Encapsulation could be supported by an additional interface concept to allow for the invocation of methods. Polymorphism would allow to dynamically activate different entities during run time. A respective approach has been proposed [8],[9] as part of the US RASSP program.

It is an algorithmic model which for the first time allows a designer to specify a circuit without any forecast of the later used hardware components. It is not the goal of synthesis to map the algorithmic specification into an isomorphic hardware structure; the algorithm is rather a possible example solution for the design problem. Hence the subjects of the intellectual design process from the algorithmic level upward are not existing hardware components, but rather abstract concepts for solving the design problem. This point of view requires a radical change in paradigms, since now a real top-down design process is possible. The objects of the design process are now abstract processes running on a simulation engine. Each process is characterized by its state space and its I/O behaviour. The later hardware structure is not implied by the structure of the algorithmic specification, but may be chosen freely as long as the functional correctness is maintained. This freedom is currently not yet fully exploited by high-level synthesis tools, but could lead to better manually optimized results even today.

An object-orientation of this view of the design process would require to support object-oriented modelling of data structures. Objects are abstract containers, which can accommodate extensible (inheritance, class hierarchy) data types. The values of these containers could be manipulated via method calls (messages). An encapsulation could be achieved by an exclusive access via visible methods [12].

In conclusion one can state that there are at least two different paradigms of object-orientation in hardware design. The evolutionary design is based on existing physical or synthesizable design objects and suggests an encapsulation based on structure [8],[9]. The implementation is the reusable intellectual property right here. The shift in paradigms required from the designer is moderate, because he/she can stay with the traditional bottom-up way of designing. If, however, a real top-down design is envisaged, which does not utilise existing physical or synthesizable hardware components, a shift in paradigm towards a data type driven object-orientation provides more abstract modelling power. Here abstract algorithms and data concepts for solving the design problem are the reusable intellectual property rights. An increase in productivity is expected during the specification and high level design phase. Due to the consistency between this paradigm and the object-orientation as used in software-engineering, a good support of hardware/software codesign is expected. A respective concept has been developed by Oldenburg University and OFFIS [12].

4. Acceptance of new design techniques

Introducing new design techniques and CAD tools only seldom lead to an instantaneous increase in designer productivity; in contrast, about 70% of the designers expect a temporary decrease of productivity [1]. The extent and duration of this decrease compared to the expected medium and long term increase are important parameters in the decision process whether to introduce changes in the design process.

In many cases a compromise between a smooth transition and an optimal final goal is sought. An example is the success of *C++* compared to *Smalltalk*. Smalltalk had been developed and implemented in a research environment. Smalltalk may be regarded as the mother of object-oriented programming languages. It was a complete new development implementing the concepts of object-orientation in their purest form. Still Smalltalk could not compete with C++ in software industry. A reason is certainly the close relationship between the *industry standard C* and C++ which promises low transition cost and compatibility. This statement is supported by a survey of the US EDA-user organisation USE/DA [1] according to which 75% of the designers mention insufficient compliance with standardized hardware description languages as reason for changing a CAD vendor.

Several languages exist in the domain of hardware specification which partly support object-oriented modelling [6]. One of them is SDL-93 [13] which is CCITT standard and used in the telecommunication domain. Because of this limited user community it is insufficiently supported by tools. In particular a link to further synthesis steps is missing. Hence SDL only supports the specification phase. In the course of the design process the designer

has to use other languages. An automatic translation of SDL models into VHDL has to fail due to some concepts of SDL, e.g. the dynamic creation of processes [6]. Existing translation systems, like the one described in [14], therefore support only SDL-subsets. This example demonstrates that just the standardization itself does not provide a sufficient user base which finances the development and maintenance of the required tool set.

Hence an object-oriented hardware description language needs to be standardized and target a large user community to be successful. A standardization of object-oriented extensions can be possible if they are compatible with the fundamental concepts of VHDL, allow for an upward compatibility of models and the cost introduction and tool development is less than the expected gain in productivity.

Additionally it is important to understand that an object-oriented modelling language on itself can only provide a limited gain. Its full power can only be utilised if linked with an adapted design methodology, as proposed in the ESPRIT project 8641, INSYDE [10].

Both mentioned proposals [8],[9],[12] use constructs of Ada [11] and have links to the analysis method by Booch [7]. A detailed comparison of both proposals has been published in [15]. They are currently under discussion in the respective standardization committee[1].

5. Summary and conclusion

This paper has analysed the suitability of object-oriented modelling techniques as a mean for productivity increases in the domain of hardware system design. The modelling cost during different phases of design has been investigated. The largest possible gain is expected in the initial specification and debugging phase of complex systems as well as in the re-use of subsystems.

As an important decision criterion for the kind of language extensions the subject of object-orientation has been elaborated. One kind of objects could be physically existing objects, which could be modelled in an abstract, encapsulated and reusable way and hence support a bottom-up design approach. On the other hand, object-orientation could be applied to abstract concepts if these are synthesizable down to the circuit level. In the first case the re-used intellectual property right is the physical implementation and an increase in productivity is to be expected during the implementation phase. In the second case, concepts based on abstract data types are subject of re-use and a more efficient specification and high level design phase is expected.

Both kinds support better team work and isolation of design task by encapsulation. Further they help to solve the administration of model data bases due to their inheritance concepts. They are currently discussed by the IEEE VHDL standardization committee. For their acceptance and success the standardization and the support by a large designer community is essential. The ESPRIT OMI project 20616, REQUEST, is going to participate in this discussion in order to represent the partners' requirements. Furthermore prototype tools will be developed as part of REQUEST to support and assess the new design methodology and language.

1. DASC Study Group on OO Extensions to VHDL

6. Literature

[1] USE/DA (User Society for Electronic Design Automation). *Results of the 1993 USE/DA Standards Survey.* Feb., 1994

[2] R. A. Bergamaschi. Productivity Issues in High-Level Design: Are Tools Solving the Real Problems? 32nd *Design Automation Conference*, San Francisco, 1995

[3] V. Preis, R. Henftling, M. Schütz, S. März-Rössel. A Reuse Scenario for the VHDL-Based Hardware Design Flow. *Proceedings of the EURO-DAC '95 with EURO-VHDL '95*. IEEE Computer Society Press, 1995

[4] E. Girczyc, S. Carlson. Increasing Design Quality and Engineering Productivity through Design Re-use. 30th *Design Automation Conference*, Dallas, 1993

[5] P. Kission, H. Ding, A. A. Jerraya. VHDL Based Design Methodology for Hierarchy and Component Re-Use. *Proceedings of the EURO-DAC '95 with EURO-VHDL '95*. IEEE Computer Society Press, 1995

[6] G. Schumacher, W. Nebel. Survey on Languages for Object Oriented Hardware Design Methodologies. *Current Issues in Electronic Modeling*, Issue 1, Kluwer Academic Press , 1995

[7] G. Booch. *Object oriented design: with applications.* Redwood City, CA, Benjamin/Cummings, 1991

[8] B. M. Covnot, D. W. Hurst, S. Swamy. OO-VHDL An Object Oriented VHDL. *Proceedings of the VHDL International User's Forum*, 1994

[9] S. Swamy, A. Molin, B. M. Covnot. OO-VHDL Extensions to VHDL. *IEEE Computer*, Oct. 1995

[10] E. Holz, D. Witaszek, M. Wasowski, S. Lau, J. Fischer, P. Roques, L. Cuypers, V. Mariatos, N. Kyrloglou. *INSYDE Integrated Methods for Evolving System Design ESPRIT: P8641. Technology Assessment. Report.* Alcatel Bell Telephone, Dublin City University, Humboldt Universität zu Berlin, Intracom S.A., Verilog S.A., Vrije Universiteit Brussel

[11] ISO/IEC 8652:1995(E) *Ada Reference Manual*, Language and Standard Libraries, Version 6.0 1994

[12] G. Schumacher, W. Nebel. 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

[13] CCITT *Revised Recommendation Z.100 CCITT Specification and Description Language (SDL)*, 1992

[14] W. Glunz. *Hardware-Entwurf auf abstrakten Ebenen unter Verwendung von Methoden aus dem Software-Entwurf.* PhD thesis. (in german) Paderborn/München, 1994

[15] W. Nebel, G. Schumacher. Konzepte objektorientierter Hardware-Modellierung. Invited talk: 2. GI/ITG/GME-Workshop *"Hardwarebeschreibungssprachen und Modellierungsparadigmen"*, Darmstadt, Feb. 15-16, 1996