Extending the SystemC Synthesis Subset by Object-Oriented Features

Eike Grimpe
OFFIS Research Institute
Escherweg 2
26121 Oldenburg, Germany
grimpe@offis.de

Frank Oppenheimer
OFFIS Research Institute
Escherweg 2
26121 Oldenburg, Germany
oppenheimer@offis.de

ABSTRACT

In this article we present an approach to object-oriented hardware design and synthesis based on SystemC. We will give an introduction to an extended SystemC synthesis subset which we propose, and, in particular, its object-oriented features. We will also briefly outline our basic synthesis concepts for object-oriented hardware specifications. Finally we will present some examples for the application of the extended synthesis subset, which are directly processable by a first synthesis tool prototype which we have developed for this purpose.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Automatic synthesis, Hardware description languages*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*VLSI*; C.0 [**General**]: *System specification methodology*; C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*

General Terms

Design, Languages

Keywords

SystemC, C/C++ based design, hardware description language, hardware synthesis, system level design, object-orientation, high-level synthesis

1. INTRODUCTION

One of the most serious challenges in embedded system design today is the growing design productivity gap. Driven by the enormous technological advances in semiconductor manufacturing and by ever increasing demands for extended functionality the complexity of embedded systems is rapidly growing. Unfortunately the designer's productivity does not grow at the same rate causing a growing gap between what can be technically realized on a single chip, and what can be effectively mastered by a design staff of reasonable size, in reasonable time.

Most probably, there will not be a sole solution to this problem, but rather a combination of different ones, including, for example, IP-centric modelling, application specific processor synthesis, a growing portion of software, and so on. However, we believe that increasing the level of abstraction which can be directly applied to hardware design is another important building block. A lesson which can be learned from software engineering is that the object-oriented modelling paradigm can help a lot in successfully handling systems of great complexity. Additionally, a convergence between hardware and software modelling paradigms is quite desirable for making a further step towards real system level design. Most embedded systems today consist of hardware as well as of software, and a unified modelling approach would allow one to make the decision on hardware/software partitioning later in the design process, maybe even automatically.

For this reason the goal of our research work is the development of design methodologies, languages, and tools which support hardware synthesis from object-oriented specifications. Existing hardware design languages, such as VHDL and Verilog, do not support object-oriented modelling, and lack respective language concepts. The still emerging system level design language SystemC™ [2, 7] would principally allow for applying object-oriented modelling, since it is in fact a C++ class library which provides the possibility of modelling and simulating hardware based on C/C++. But existing synthesis tools which are able to process SystemC specifications, such as the CoCentric™SystemC Compiler[1] [13, 14], only support a synthesis subset that is widely equivalent to common RT/Behavioral level VHDL and Verilog synthesis subsets, excluding almost all object-oriented C++ features. We try to overcome these limitations with new synthesis techniques, and also with some additions to SystemC which provide some object-oriented concepts in a way which better suits the requirements from hardware synthesis.

The remainder of this article will first give a short overview of related work. Section 3 will very briefly define the terms structural and data type based object-orientation in order to allow for a better classification of the approach presented in

[1]In fact this is the only working synthesis tool at the moment, the authors of this paper are aware of, which is able to process SystemC

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

this work. Section 4 will introduce and discuss our proposed extended SystemC synthesis subset. Section 5 outlines our basics synthesis concepts for object-oriented language constructs, and section 6 will present some examples illustrating the application of the extended subset. The article will close with the conclusions, and an outlook on future work.

2. RELATED WORK

Over the last years there have already been several approaches on object-oriented hardware design. Some of these approaches are based on object-oriented programming languages and augment them by capabilities for hardware modelling [4, 3, 15], whilst other approaches are based on existing hardware description languages and augment them by object-oriented language features [12, 11, 1]. None of these approaches reached a general breakthrough so far, because they either primarily focus on simulation rather than on synthesis, lack sufficient tool support, or, like SystemC, make only use of an object-oriented programming language in order to add hardware modelling capabilities, but basically do not allow for object-oriented modelling. However, especially the VHDL based approaches [9, 11, 1] have strongly motivated and influenced our work.

3. STRUCTURAL VS. DATA TYPE BASED OBJECT-ORIENTATION

We distinguish basically two principle approaches to object-oriented hardware design. Structural approaches model whole hardware components/entities as concurrent objects. That means, for instance, that components like multiplexer, arithmetic logical units (ALU), decoders, registers, and register files are modelled as self-contained objects. Although this modelling style seems to be quite natural at first sight, it raises various problems. The combination of parallelism and object-oriented modelling poses some fundamental problems that are well known and discussed in object-oriented concurrent programming, for several years now. For example, the fundamental object-oriented concept of inheritance can only be hardly combined with the concept of concurrent, objects, i.e., objects with an own thread of control. This problem is known as so called *inheritance anomaly* [5]. Up to now, the authors of this article are not aware of a general solution to this problem.

Data type based approaches make use of object-orientation as a way to create new user defined data types, but not structural objects, as used from programming languages, such as C++ and Java. Objects in this sense do not have an own thread of control, and they are not going to be independent entities as discussed above. First class objects are user defined data objects, such as data packages, data containers, computational elements, like complex numbers, vectors, matrices or filters, etcetera. These elements are used within a sequential context, usually a process, and may move from place to place in a design, but do not determine the structure of a system.

Our approach focuses on data type based object-oriented modelling for mainly two reasons; first, because of the problems of the structural approaches as mentioned above, and second, because it can be applied relatively easy, and in a straightforward way, for extending the limited SystemC/C++ subset which is supported for synthesis by existing tools. The application of a kind of structural object-orientation is also possible, based on so called global objects (refer to section 4.3), but only in a very limited way, thus avoiding most of the problems of object-oriented concurrent programming.

4. EXTENDING THE SYSTEMC SYNTHESIS SUBSET

The modelling language SystemC is gaining increasing importance as system level design language for several reasons. First of all it allows one to model hardware based on C/C++, since it is in fact a C++ class library comprising hardware modelling and simulation features. C/C++ is presumably the most widespread and known programming language most designers are familiar with even more likely than with any hardware description language, such as VHDL or Verilog. And C/C++ is supported by a wide range of established tools and development environments, a lot of them even available as freeware. A SystemC design can just be compiled with most common C++ compilers, and then be simulated by running the resulting executable without requiring an external simulator. Starting the design process with a 'golden model' written in C/C++ is a common practice in embedded system design. SystemC allows one to keep this language whilst refining a design down to a cycle accurate or even synthesizable level. Moreover modelling at different levels of abstraction and new design methodologies, such as *transaction level modelling*, provide for the possibility of a quick and flexible design space exploration.

Although SystemC does not limit the use of C++ in principle, existing synthesis tools for SystemC do not support object-oriented features, as mentioned in the introduction. But by applying some additional synthesis techniques, as being outlined in section 5, this restricted synthesis subset can be extended by a wide range of object-oriented C++ features which are discussed in the following.

4.1 Classes and Co.

The basic language feature of C++ that allows for object-oriented modelling is its class concept in combination with inheritance. Therefore our extended synthesis subset includes most of the class related language constructs provided by C++. In particular, this means the ability to:

- declare classes;
- create new classes by means of inheritance from already existing classes;
- derive classes from multiple and virtual parent classes;
- declare data members of scalar and complex type, including class types and array types;
- declare member functions and operators;
- redefine member functions and data members in derived classes;
- declare constructor methods;
- declare class templates with scalar and type parameters.

As being common practice in C++, classes can be used as types for objects in a design, for instance, as type for a variable, a signal, or port, as a return type for a function, or as a type for a formal parameter. A class type can just be used wherever another type can be used. Data members and member functions can also be used and accessed in the same way as in C++.

Supporting these features should provide for the possibility of declaring and using classes and objects in basically the same way as usual in C++, but there are, of course, also some restrictions and limitations with respect to hardware modelling and synthesis. For instance, dynamic object creation, and destruction by means of the `new` and `delete` operators are excluded from the synthesis subset for obvious reasons. Pointer variables, or, more precisely, pointer arithmetic, is also excluded. We believe that pointer synthesis, although possible in principle, tends towards producing inefficient hardware, in terms of area and speed, but, at the same time, offers only very little modelling benefit in the absence of dynamic memory allocation.

4.2 Polymorphism

Polymorphism is a modelling concept which, simply speaking, allows one to handle objects, or references to objects, of different classes but with common features uniformly. Objects of different classes may provide the same abstraction of an operation—which means with the same signature—but with different implementations. An operation that is requested on a reference that may refer to objects of different classes is dynamically bound[2] to a concrete implementation at runtime, dependent on the class of the object actually being referenced. Whoever requests this operation does not have to explicitly take care about that. A classical example from software engineering for the application of polymorphism is a class hierarchy of geometrical objects, like `Circle` and `Square`, being derived from a common base class, all providing a `draw()`-method. If an instance of a geometrical object should be drawn to a screen just this `draw()`-method has to be invoked without having to know the concrete type of the geometrical object. A runtime system will automatically bind the correct method implementation to the call.

There is also a wide range of potential applications for polymorphism in hardware. For instance, different formats of data packages, or datagrams, may differ in several aspects but may share the same address format. A hardware component with the purpose to route data packages could handle packages of different formats, even dynamically at runtime, since it only would have to deal with the address, whose format is always the same. Another example are different kinds of filters which could be dynamically exchanged at runtime dependent on actual needs, e.g., for best fitting varying kinds of data streams. Further examples include instructions and operations that could be grouped according to common features. For instance, a store instruction usually has a source and a destination, no matter, if it operates on registers or memory addresses, and whether it includes immediate operands or not. Likewise, binary arithmetical operations, as illustrated by Figure 1, always have two operands and return a result on execution. A concrete example for the application of these operations is given in section 6.

[2]Regarding runtime, or dynamic, polymorphism.

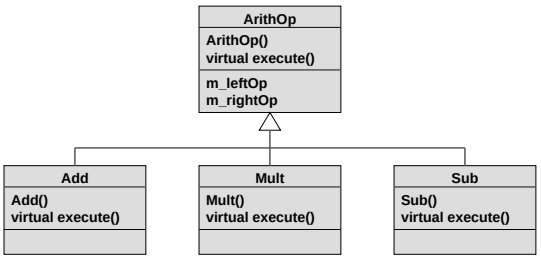

Figure 1: Class hierarchy of arithmetic operations

The polymorphism mechanism provided by C++, and therefore by SystemC, too, is based on the use of pointers. A pointer may reference instances of different classes at runtime, and thus methods invoked on a pointer are dynamically dispatched. As explained in the previous section, pointers are excluded from our synthesis subset, making an alternative polymorphism realization necessary.

What we propose as alternative, and what is also supported by our synthesis tool prototype, is a *tagged object* approach. A tagged object—we call this also a polymorphic object—is a self-contained object, with an own state space, which means it is not just a reference to an object, like a pointer. This makes reference resolution for a polymorphic object at runtime obsolete. Only its actual class must be determinable at runtime. In the synthesised hardware the actual class of a polymorphic object is tracked by an artificial attribute, called *tag*. More details on this are given in section 5. The important point for modelling is, that a polymorphic object provides for dynamic binding of operations, and that it behaves widely similar to a pointer in C++, for instance, in respect of assignment rules. Making use of polymorphic objects requires for including an additional C++ class library, called 'OOHWLib', which is freely available. An application example is given in section 6.

4.3 Global Objects

Communication modelling is one of the key aspects in the design of complex embedded systems. Methodologies and techniques that would help to improve the modelling of communication between subcomponents could also significantly improve the whole design process. For this reason SystemC provides the concept of channels for communication. A channel can be used to abstract from the details of a certain form of communication—for instance, a certain protocol—, and allows processes to communicate and exchange data with each other, based on method calls instead of signals. This concept already has led to a new kind of design methodology, called *transaction level*, or *transaction based*, modelling, actually being intensively discussed in the SystemC community [10].

Indeed the channel concept provides for modelling communication at a higher level of abstraction, and to separate communication and behavior more cleanly. But unfortunately, channels do not generally possess a clear synthesis semantics. And, although being intended for communication between concurrent components, they do not posses built-in mechanisms for handling concurrent accesses. This makes channels quite useful for creating un-timed models and for simulation but not for synthesis.

For this reason we offer an alternative concept to SystemC channels, based on so called global objects. A global object is, in principle, an object which is declared as member of a SystemC module, like a port or signal. Therefore it is visible and accessible by all processes declared in the same module. Like any other object a global object may specify a set of methods which forms its interface. This allows processes to communicate and exchange data via global objects based on method calls, as illustrated in Figure 2, similar to SystemC's channels. Like ports and signals, global objects which are located in different modules can be bound to each other, thus making communication throughout a hierarchy of modules possible.

Figure 2: Method based communication via global object

In contrast to channels, a global object possesses built-in mechanisms for handling concurrent accesses, a clear synthesis semantics, and it exhibits a timed behavior during simulation. Handling concurrent accesses is basically realised by means of a scheduler which has to be specified for each global object by the user. The scheduler determines which client process is granted to access the global object in the case of concurrent requests. All other requesting clients are blocked meanwhile. As a consequence accesses to a global object are mutual exclusive. In addition this is supported by a guard mechanism which allows one to associate a Boolean expression—the so called *guard*—with a member function. Clients requesting an operation whose guard is 'false' at that moment are ignored for scheduling and are blocked. The timed behavior of global objects gives the designer an early and realistic impression on the temporal behavior of the modelled system during simulation even before preforming synthesis.

In order to declare and to use global objects in a design, the user has to include the same additional class library, that also provides the polymorphic objects discussed in the previous section.

5. SYNTHESIS BASICS

In this section we will give a very brief overview of our basic synthesis concepts. More details can be found in [8] which builds the foundation of our work.

The key of the synthesis concept we apply to object-oriented hardware specifications forms a mapping of objects to bit vectors. Data members of an object are represented by slices of the generated bit vector. Therefore, the width of a bit vector representing an object is given by the sum of the bit widths of all individual data members of the object. For instance, for a Boolean member 1 bit is added, and for an arbitrary sized integer (`sc_bigint<>`) the respective number of bits which were used for its declaration. If a data member is itself an object, it is transformed into a bit vector first and then contributes to the size of its super object. Wherever a class type was used before, it is replaced by a bit vector of appropriate size during synthesis. Access to data members of an object is replaced with access to the appropriate slice of the bit vector representing the original object. Member functions, also including constructors and overloaded operators, are transformed into ordinary non-member functions which operate on the synthesised bit vectors instead of the original objects.

Bit vectors representing polymorphic objects are chosen big enough to store objects of different classes, which basically means as big as the biggest object ever being assigned. Additionally, an artificial attribute—the tag mention in section 4.2—is added to the state space of a polymorphic object allowing for tracking its actual class. With the ability to dynamically determine the class of a polymorphic object by means of the tag, dynamic binding of operations can be easily realised, for instance, by means of a set of 'if statements', or a 'switch statement', which select from different implementations dependent on the current value of the tag.

From a global object a set of synchronous and asynchronous processes is generated which implement the scheduling of concurrent accesses, the evaluation of the guard conditions, the operations provided by a global object, and all the necessary multiplexing. The communication between a global object and its client processes is mapped to a signal based communication with a fixed handshake protocol.

5.1 The ODETTE Synthesizer

The ODETTE synthesizer is a synthesis tool prototype which was developed—and which is still under development—in the course of the European project ODETTE [6]. It is able to process the extended SystemC/C++ subset which is described in the previous section. Starting from a SystemC description, which is based on this subset, the synthesis tool generates behavioral equivalent SystemC or VHDL code that can be further processed with existing tools and technologies. As shown in Figure 3 the tool is settled on top of existing tool chains and does neither perform behavioral synthesis nor logic synthesis at the moment. It just maps object-oriented language elements in the input specification to behavioral equivalent lower lever constructs, according to the basic synthesis concepts outlined in the previous section.

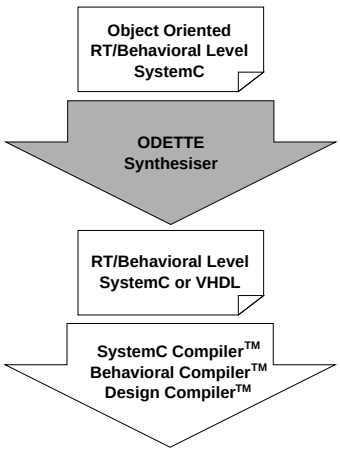

Figure 3: Synthesis flow

The synthesis tool prototype has now reached a state where it provides for the possibility of performing concrete

modelling and synthesis experiments. That means that it effectively supports all the features that are listed in section 4, and that it allows one to start the automated synthesis flow from a higher level of abstraction. But it is still a prototype offering a high potential for improvements. In particular, more sophisticated optimization techniques could be integrated which aim at producing better results in terms of circuit area and speed.

6. EXAMPLES

The following code excerpts will demonstrate the application of some higher level language concepts, namely global objects and polymorphic objects, included in our extended SystemC synthesis subset. The presented code is directly processable with the ODETTE synthesis tool prototype without any need for further manual refinement. The generated output is a SystemC or VHDL specification which can be fed into off-the-shelf synthesis tools, such as the CoCentric SystemC Compiler, or the Synopsys Design Compiler™.

In the first example a classical producer/consumer system is modelled which is also often used as an example for demonstrating the benefits of channels in SystemC. Producer and consumer are exchanging data via a bounded buffer, which represents a shared resource (see also Figure 2). In 'plain' SystemC the bounded buffer would be modelled as a channel. According to our approach it is implemented as a global object now combining high level modelling with automated synthesis.

```
#include "systemc.h"
#include "oohwlib.h"

SC_MODULE(ProdCons) {
  sc_in< bool > clock;
  sc_in< bool > reset;
  sc_out< ElementType > output;

  // Declaration of a global object with scheduler 'RoundRobin'
  // and user defined class template 'FIFO<>':
  GlobalObject< RoundRobin, FIFO< int, 12 > >
    sharedBuffer;

  void producerOdd() {
    int val;
    sharedBuffer.reset();
    sharedBuffer.subscribe();
    val = 1;
    wait();
    while( true ) {
      // Blocking global method call to 'sharedBuffer':
      GLOBAL_PROCEDURE_CALL( sharedBuffer, put( val ) );
      val += 2;
      wait();
    }
  }

  void producerEven() {...}

  void consumer() {...}

  SC_CTOR(top)
  {
    SC_CTHREAD( producerOdd, clock.pos() );
    watching( reset.delayed() == true );

    SC_CTHREAD( producerEven, clock.pos() );
    watching( reset.delayed() == true );

    SC_CTHREAD( consumer, clock.pos() );
    watching( reset.delayed() == true );
  }
};
```

Figure 4: Using a global object

Three client processes are connected to the global object `sharedBuffer`; two producers, one producing odd numbers, the other producing even numbers, and a consumer which reads elements from the buffer and writes them to an output port. In the same way any further client process could be connected. In a real world application instead of numbers any kind of data could be exchanged via the buffer, e.g., data packages, frames, or samples.

The following code shows an excerpt from the implementation of class template `FIFO` that was used in the above example. In principle it is an ordinary C++ class declaring some data members and member functions. Modelling it as a template which is parameterized with the element type and the size of the buffer makes it possible to use it very flexible. What makes the implementation different from an implementation in plain C++ is the declaration of some methods as guarded methods. But this does not prevent the class from being used like any other class, since the guard mechanism is only invoked on global objects (cf. 4.3).

```
#include "oohwlib.h"

template< class Type, unsigned int SIZE >
class FIFO {

public:
  FIFO() {
    reset();
  }

  GUARDED_METHOD( void,                    // return type
                  put( const Type &t ), // signatue
                  !isFull() ) {            // guard
    m_buffer[m_bottom] = t;
    m_bottom = nextIndex( m_bottom );
    m_empty = false;
    m_full = ( m_bottom == m_top );
    m_numberOfEntries = m_numberOfEntries + 1;
  }

  GUARDED_METHOD( void, reset(), true ) {...}
  GUARDED_METHOD( Type, get(), !isEmpty() ) {...}
  GUARDED_METHOD( void, remove(), !isEmpty() ) {...}
  GUARDED_METHOD( bool, isFull() const, true ) {...}
  GUARDED_METHOD( bool, isEmpty() const, true ) {...}

protected:
  ... // further member declarations
};
```

Figure 5: Declaration of class FIFO

The next example picks up the polymorphic arithmetic operations illustrated in Figure 1. The `execute` method is dynamically dispatched which means that one does not have to take care about the concrete type of an operation being sent via input port `operation`, as demonstrated in process `behavior`. Dependent on the actual type of the operation always the correct implementation of `execute` will be invoked. Like the previous example, the presented code can be directly processed by the ODETTE synthesiser. Note, that this is a very simplified example due to spacial limitations in this article.

```
#include "systemc.h"
#include "oohwlib.h"

class ArithOp {
  // 'tag' the class for polymorphic usage:
  POLYMORPHIC( ArithOp )
public:
  ArithOp( const int leftOp, const int rightOp ) :
        m_leftOp( leftOp ), m_rightOp ( rightOp ) {
```

```
  }

  virtual void execute( int &result ) {
    // nothing to do here. Will be implemented in derived classes.
  }

protected:
  int m_leftOp;
  int m_rightOp;
};

class Mult : public ArithOp {
  // 'tag' the class:
  POLYMORPHIC( Mult )
public:
  virtual void execute( int &result ) {
    return( m_leftOp * m_rightOp );
  }
};

class Add : public ArithOp {
  // 'tag' the class:
  POLYMORPHIC( Add )
public:
  virtual void execute( int &result ) {
    return( m_leftOp + m_rightOp );
  }
};

SC_MODULE( PolyALU ) {
  sc_in< bool > clock;
  sc_in< bool > reset;
  sc_out< int > acc;
  // Declaration of a 'polymorphic' port:
  sc_in< PolyObject< ArithOp > > operation;

  void behavior() {
    PolyObject< ArithOp > localOp;
    int result;
    wait();
    while( true ) {
      localOp = operation.read();
      opResult = acc.read();
      // The following function call is dynamically dispatched
      localOp->execute( result );
      acc.write( result );
      wait();
    }
  }

  SC_CTOR( PolyALU ) {
    SC_CTHREAD( behavior, clock.pos() );
    watching( reset.delayed() == true );
  }
};
```

Figure 6: Applying polymorphism

7. CONCLUSIONS AND OUTLOOK

In this paper we have illustrated how the limited SystemC synthesis subset which is supported by existing synthesis tools can be simply extended by most object-oriented C++ features, and we have proposed an appropriate synthesis technique for this purpose. We have also discussed how hardware design can benefit from the application of object-orientation modelling techniques. We have further presented a tool prototype which is able to process the proposed extended synthesis subset by performing the illustrated synthesis techniques, and which generates SystemC or VHDL specifications that are further processable with existing off-the-shelf tools.

Future work will mainly consist in improving the synthesis tool prototype. One major focus will be the integration of more sophisticated optimisation techniques and strategies in order to produce better results in terms of circuit area and speed. Another focus will be the further extension of the supported synthesis subset, and, in the mid- or long-term, the integration of concepts from behavioral synthesis.

8. REFERENCES

[1] P. Ashenden, P. Wilsey, and D. Martin. Suave: Object-oriented and genericity extensions to vhdl for high-level modeling. In *Proceedings of Forum on Design Languages (FDL98)*, pages 109–118, September 1998.

[2] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[3] T. Kuhn, T. Oppold, C. Schulz-Key, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai. Object oriented hardware synthesis and verification. In *ISSS'01*, pages 189–194, October 2001.

[4] T. Kuhn, W. Rosenstiel, and U. Kebschull. Object oriented hardware modeling and simulation based on java. In *International Workshop on IP Based Synthesis and System Design, Grenoble, France*, 1998.

[5] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[6] ODETTE. *Object-oriented co-DEsign and functional Test TEchniques*. http://odette.offis.de.

[7] Open SystemC Initiative. *SystemC Version 2.0 User's Guide. Update for SystemC 2.0.1*, 2002.

[8] M. Radetzki. *Synthesis of Digital Circuits from Object-Oriented Specifications*. PhD thesis, University of Oldenburg, 2000.

[9] M. Radetzki, W. Putzke-Röming, and W. Nebel. Objective vhdl: The object-oriented approach to hardware reuse. In J.-Y. Roger, B. Stanford-Smith, and P. T. Kidd, editors, *Advances in Information Technologies: The Business Challenge*, 1998.

[10] H.-J. Schlebusch, G. Smith, D. Sciuto, D. Gajski, C. Mielenz, C. K. Lennard, F. Ghenassia, S. Swan, and J. Kunkel. Transaction based design: Another buzzword or the solution to a design problem? In *Proceedings of DATE'03*, 2003.

[11] G. Schumacher. *Object-Oriented Hardware Specification and Design with a Language Extension to VHDL*. PhD thesis, University of Oldenburg, 1999.

[12] S. Swamy, A. Molin, and B. Covnot. Oo-vhdl object-oriented extensions to vhdl. *IEEE Computer*, 28(10):18–26, October 1995.

[13] Synopsys, Inc. *CoCentric(R) SystemC Compiler Behavioral Modeling Guide*, 2002.

[14] Synopsys, Inc. *CoCentric(R) SystemC Compiler RTL User and Modeling Guide*, 2002.

[15] S. Vernalde, P. Schaumont, and I. Bolsens. An object oriented programming approach for hardware design. In *IEEE Computer Society Workshop on VLSI'99*, Orlando, April 1998.