Hardware Reuse at the Behavioral Level

Patrick Schaumont Radim Cmar Serge Vernalde Marc Engels Ivo Bolsens
IMEC vzw, Kapeldreef 75, B-3001 Leuven Belgium

Abstract

Standard interfaces for hardware reuse are currently defined at the structural level. In contrast to this, our contribution defines the reuse interface at the behavioral register-transfer (RT) level. This promotes direct reuse of functionality and avoids the integration problems of structural reuse. We present an object oriented reuse interface in C++ and show the use of it within two real-life designs.

1 Introduction

The rush forward of digital implementation technology faces contemporary chip designers with ever increasing design complexities. This makes the ability to reuse components in a system an essential design skill. Examples of such components are embedded cores or complex random logic blocks. The VSI (Virtual Socket Interface) Alliance is an industry-backed organization that targets the requirements and standards definition for component reuse.

The established view on reuse is focused at the structural level. A component is made reusable by matching it to a standard interface. This interface defines input/output signals, their timing relationship etc. Such an interface allows hiding of the detailed design of a component as intellectual property (IP) of a designer, and yet makes the component available for reuse.

The reuse of hardware components at the structural level is not without problems, because of the following reasons:

- Reuse is in the first place a matter of reusing functionality, not structure. It happens often that a component can be 'almost' reused, but requires additional encapsulation to match the right behavior.
- Structural reuse seals the behavior of a component in a closed box behind the reuse interface. As a result, the reused behavior can only be manipulated indirectly through this interface.
- Current hardware development environments are good in capturing, simulation and synthesis of hardware components. They do however a bad job in *manipulating* the same descriptions. As an example, VHDL defines a component as an entity with a well defined port set. It is not possible to strip the ports of an entity depending on some external design condition.

As shown in Table 1, the impact of reusing structure instead of functionality can be substantial. The table collects some statistics for a DECT transceiver we recently designed. It lists the number of blocks in the chip (`total`). An amount of those (`prog`) require datapath register access, which is embodied in a per-block programming interface. Next, the total RT-VHDL line count is shown, first without this programming function (`wo/prog`) and next including it (`w/prog`). As this function is parameterized by the number of datapath registers per block, it cannot be obtained through simple instantiation. Therefore, introduction of this per-block function requires significant RT coding overhead.

This situation has been recognized by other authors as a 'Silicon Ceiling' [7]. Research solutions have been either to encapsulate VHDL within an advanced design environment [6] or else to extend the semantics of VHDL itself [1, 2].

Being faced with structural reuse problems in several recent demonstrator designs, we developed a hardware reuse mechanism at the more abstract behavioral RT-level. Rather than tackling this problem at the VHDL level, we used a C++ based development environment and an object oriented RT model [8, 5]. The use of C++ has been proven to be an adequate container for modeling and simulation of parallel hardware systems [12, 4]. Our environment includes the capturing and the simulation of a digital hardware system, and also contains a code generator that translates the C++ description to synthesizable VHDL. In addition, it supports HDL testbench generation that allows to verify the C++ description against the hardware synthesis results.

This contribution places the emphasis on the hardware reuse mechanism. Section 2 presents two motivational examples that clearly state the behavioral reuse problem. In section 3, our C++ class hierarchy for the description of digital hardware is presented. Section 4 will define a behavioral reuse mechanism that builds upon this class hierarchy. Section 5 picks up the two motivational examples again, and applies the reuse method on it. Finally, a summary of the contributions is given in section 6.

Block count		RT VHDL linecount	
total	prog	wo/prog	w/prog
25	23	17K	28K

Table 1: RT line counts for DECT design

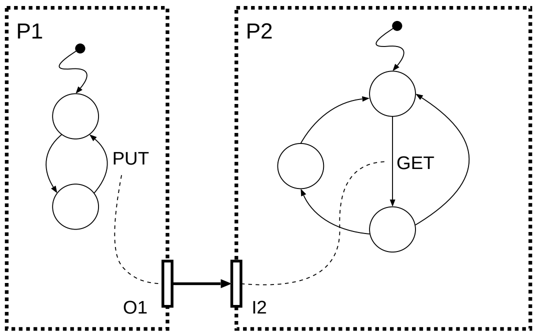


Figure 1: Communicating Processors

2 Problem Statement

In this section, two hardware design problems are introduced to sketch the problems of reuse. The first one is an interblock synchronization interface, while the second one is a programming interface for ASICs. Both of the design problems originate from the close interaction between the internal behavior of a block and the external world [10, 9].

2.1 Interblock synchronization

Figure 1 shows a simple case of two communicating processors, `P1` and `P2`. Each of the processor's behavior is described through a finite state machine (FSM). The nodes indicate execution states, while the edges correspond to state transitions for every clock cycle of data processing. Each of the FSMs thus represents the schedule of an algorithm. The `GET` and `PUT` operations show at which clock cycles the processors communicate data. This shows that there is an input/output dependency between processors `P1` and `P2`. When unsynchronized, processor `P1` produces output data every second clock cycle. This data is consumed by processor `P2` with a variable schedule of two or three clock cycles. The communication of data thus introduces a synchronization requirement between `P1` and `P2` to guarantee correct operation of the system. The current practice to solve this kind of communication consistency problem is to use one of the following methods.

a) To adapt each of the processor's description such that they are always in perfect synchrony.

b) To introduce a global synchronization mechanism that forces communication synchrony.

c) To embed a universal communication protocol onto the IO ports.

When thinking in terms of reuse none of these three solutions is optimal. Cases a) and b) force designers to solve two interdependent tasks at the same time (local and global behavior), resulting in a difficult and hard-coded solution. Case c) implies the use of a universal communication mechanism which might represent an overhead at the next application.

Structural reuse becomes hard, or in the best case causes an overhead in silicon and/or timing. What we need here is a method that automatically *manipulates* the local processor behavior and adapts it to the existing communication environment. Such a method is generic, and is a good candidate for behavioral reuse.

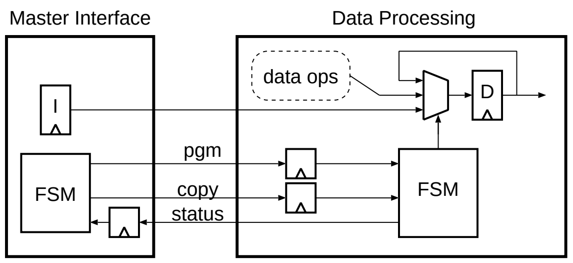


Figure 2: Programming Interface

2.2 A programming interface

The second design problem, a programming interface, is a common feature in ASICs. An example is shown in figure 2. It consists of two blocks out of a synchronous ASIC design. Only the parts relevant to the programming interface are shown. The first is a `Master Interface`. The purpose of this block is to make the data processing registers of the ASIC programmable from the outside world. The second block, `Data Processing`, is a functional component of the ASIC. This block has a local controller `FSM`, that sequences instructions to a datapath. Doing this, a digital signal processing (DSP) algorithm such as equalization can be implemented. Furthermore, this local controller also performs additional instructions, which are invoked by the master interface through `pgm` and `copy`.

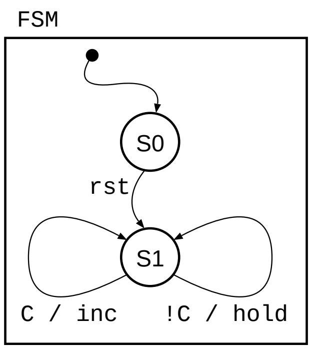
The data processing block has two modes of operation: an active mode, and a programming mode. The desired mode is set by the master interface through the value of `pgm`. The data processing block also signals which mode is currently active through a `status` bit. The data register `D` is updated when the master interface sets the `copy` bit and at the same time the data processing block is in programming mode.

A simple protocol controls the programming of the data register `D`. When a value is available in register `I`, the master interface sends a program mode request to the data processing block by setting the `pgm` bit. Depending on the real time requirements inside the data processing algorithm, the data processing block will enter the program mode some cycles later and signals this to the master interface through the `status` bit. The master interface then can update the data register `D` by setting the `copy` bit.

The design complexity of the data processing block lies in the simultaneous presence of DSP algorithm and programming protocol. As a consequence, the designer of the data processing block needs to master both a DSP algorithm schedule and a protocol. Whether the FSM is described hierarchically or not does not matter: the designer needs to think of two things at once.

In addition, using current HDL environments, it is not possible to design the DSP processing schedule of the block independently of the protocol, which degrades potential reuse possibilities.

What is really needed here is a method that allows the data processing to be designed independently from the interfacing. This will be possible by describing the programming interface as a case of behavioral reuse (Section 5.2).



```
register C;
register N;
input    IB;
output   OB;

rst:   N = 0;
       C = IB;

inc:   N = N + 1;
       C = IB;
       OB = N;

hold:  C = IB;
       OB = N;
```

Figure 3: Ones-counter behavioral RT

3 The OO-RT data model

The OO-RT model is a framework that enables behavioral reuse. In this section, the OO-RT model is explained. We start with a classic RT specification of a simple processor that counts the number of '1' bits in a bit stream. Next, an OO-RT version of the same processor is presented. The ones-counter processor contains the following elements.

- A datapath with two registers. Register `N` holds the number of '1'-bits seen after the last reset, while register `C` holds the value of the currently observed bit in the bit stream. It is assumed that the count register `N` has sufficient width to hold the maximum bit count during two subsequent reset instructions.
- A controller `FSM` that can increment, hold or reset the count register in the datapath.

Figure 3 shows a behavioral RT specification of this ones-counter. The specification consists of a Mealy-type state transition diagram, and three RT instructions `rst`, `inc` and `hold`. These correspond to the datapath actions in case of reset, observation of a '1'-bit and observation of a '0'-bit respectively.

This behavioral RT specification is now mapped into an object oriented model. The C++ specification shown below corresponds to the representation of figure 3.

```
 1: clk ck;
 2:
 3: sig C(ck,0):
 4: sig N(ck,0);
 5: sig input;
 6: sig output;
 7: bus IB;
 8: bus OB;
 9:
10: sfg rst;
11: N = 0;
12: C = input;
13: rst << in(input,IB);
14:
15: sfg inc;
16: N = N + 1;
17: C = input;
18: output = N;
19: inc << in(input,IB) << out(output,OB);
20:
21: sfg hold;
22: C = input;
23: output = N;
24: hold << in(input,IB) << out(output,OB);
25:
26: fsm ones_cnt;
27: state s0;
28: state s1;
29: ones_cnt << deflt(s0);
30: ones_cnt << s1;
31: s0 <<  always << rst  << s1;
32: s1 <<  cnd(C) << inc  << s1;
33: s1 << !cnd(C) << hold << s1;
```

The data processing is expressed in terms of `sig` objects, that represent plain signals or registers (lines 3-6). Datapath instructions such as `rst`, `inc`, and `hold` are described using the `sfg` objects. Each of these group a number of signal expressions (lines 10-24). The I/O ports of the behavior are indicated using `bus` objects (lines 7-8).

The control description of the ones-counter is captured by a direct modeling of the FSM description in figure 3. Each state of the ones-counter FSM maps into one `state` object (lines 27-28). The `fsm` object groups a number of `state` objects (lines 29-30), identifying one as the initial state (line 29).

The datapath instructions are assigned to control steps by creating FSM transitions (lines 31-33). A transition contains a source state, a transition condition, a datapath instruction to execute, and a target state.

When this C++ description executes, a hierarchy of objects is constructed. Rather than directly executing code, a data structure is created first, that subsequently can be processed by an interpreter or a code generator.

Operator overloading is used extensively to create this object hierarchy, and to relate individual objects to each other. For example, the shift operator (<<) is used to define datapath I/O (lines 13, 19, 24). It is also used to associate a `state` object with an `fsm` object (lines 29-30). Finally, it is used to define state transitions (lines 31-33). In each of those cases the shift operator is used to establish a relationship between individual objects. An `fsm` object for example retains a pointer to each `state` object it contains.

As a result of this operator-overloading strategy, the complete RT behavior of the ones-counter is captured in an object hierarchy. This hierarchy is a data structure made up out of objects that represent behavioral-RT concepts such as signals, datapath instructions, or controller states. One way to represent this object hierarchy more formally is to use a *class diagram.* Figure 4 illustrates this diagram using the Object Modeling Technique (OMT) [3]. Each rectangle indicates a class, while the arrows indicate class relations. An arrow ending in a filled circle indicates a one-to-many relation, such as the relation of a single `fsm` object to the several `state` objects it contains. An arrow starting with a diamond indicates a part-of relationship; while a plain arrow-start indicates a non-exclusive relationship.

The class diagram in figure 4 indicates that a reference to an `fsm` object is sufficient to retrieve the complete processor description as a set of interrelated objects. Each other object can be reached by following the necessary links of relationship. In the next section, a reuse mechanism will be defined that uses this object hierarchy.

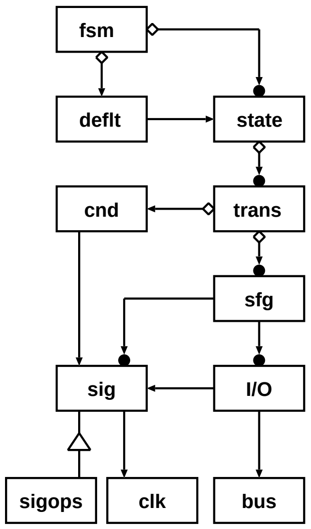


Figure 4: Class Diagram

4 Behavioral Reuse in the OO-RT Data Model

Traditional hardware reuse focuses on the reuse of prebuilt blocks. This is possible by providing these blocks with a *standard interface* (specifying port naming conventions, electrical and timing properties, etc) to connect them to the system.

The OO-RT data model promotes reuse because it constructs systems according to a standard class diagram, as shown in figure 4. This set of classes, and their relations can play the role of a standard interface. The reusable blocks then become pieces of RT behavior that are built up out of standard OO-RT objects themselves, and that interact with an existing object hierarchy. We can add new objects to an existing object hierarchy, or else modify existing relationships. Some examples of useful manipulation of the object hierarchy are:

- Changing the transition condition (`cnd` object) of a state transition (`trans` object).
- Attaching extra wait states (`state` object) and transitions to an fsm (`fsm` object), e.g. to add a synchronization capability.
- Adding extra operations (`sigops` object) to an instruction (`sfg` object) to extend it, e.g. to add overflow detection in the ones counter.

For the purpose of manipulation, the classes presented in figure 4 have methods that allow to query and/or modify their relationships towards other classes.

The method of behavioral reuse works at a finer granularity then structural reuse, and can incrementally change existing behavior. Consider for example a `state` object, which is used to represent a control state. It can be added to an existing `fsm` object in order to extend the set of control states that this `fsm` understands. New transitions can be created from existing states towards the newly created state to extend the control flow.

We found that this kind of object oriented manipulation is best organized into a new object, a *reuse object*. This new object does contain a piece of register transfer code that can be attached to existing behavior. An example of such an object is the `waitstate` object.

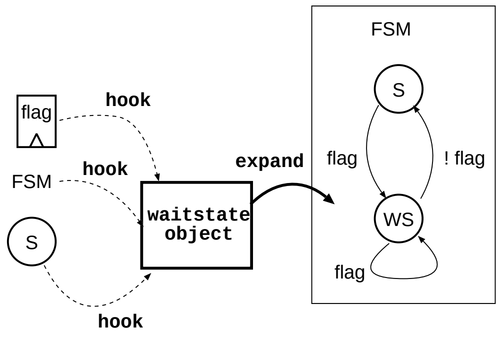


Figure 5: Manipulations by the `waitstate` object

```
class waitstate {
  state ws;
  public:
    waitstate();
    expand(fsm &f, sig &flag, state &s) {
      f   << ws;
      s   << cnd(flag)  << ws;
      ws  << cnd(flag)  << ws;
      ws  << !cnd(flag) << s;
    }
};
```

This object adds a wait state to an existing state in an FSM. Given an FSM `f`, a start state `s` and a signal `flag`, it will modify the FSM such that it includes a conditional jump from the start state to a newly created wait state `ws`. We will call the existing objects to which a reuse object is attached the *hooks*. In this case, the hooks are: an FSM `f`, a state `s` and a signal `flag`. Also, we will call the procedure that performs the manipulation the *expand* method. The effect of the `waitstate` object is shown in figure 5.

In the next section, the two design problems introduced earlier are solved with a behavioral reuse mechanism.

5 Application and Results

5.1 Interblock synchronization

The interblock synchronization is now solved as an application of behavioral reuse. Figure 6 shows a part from the example of section 2.1 as an input for reuse. We explain the synchronization solution for the case of the `PUT` instruction done by processor `P1`. This processor is connected to the processor `P2` via a communication bus object. The immediate implementation of such a bus object is simple wiring. However, in the case that the `PUT` has to be synchronized to the corresponding `GET` in processor `P2`, a `synchronizer` object comes into play. The synchronizer object will take care of merging a synchronization protocol into `P1`'s OO-RT description. In `P2`'s OO-RT description, a similar synchronizer object is used to provide a matching protocol.

Being a reuse object, the `synchronizer` needs *hooks* and an *expand* method. The hooks for this reuse object are a communication bus on one hand, and an FSM that reads/writes this communication bus on the other. Given the fsm of `P1` and the bus object that carries the PUT, the `expand()` method of the synchronizer modifies the OO-RT description of `P1` as shown on the bottom of the figure.

Several modifications take place during the expansion. First, a wait transition is inserted. In addition, new instructions are added, which provide the signaling of a syn-

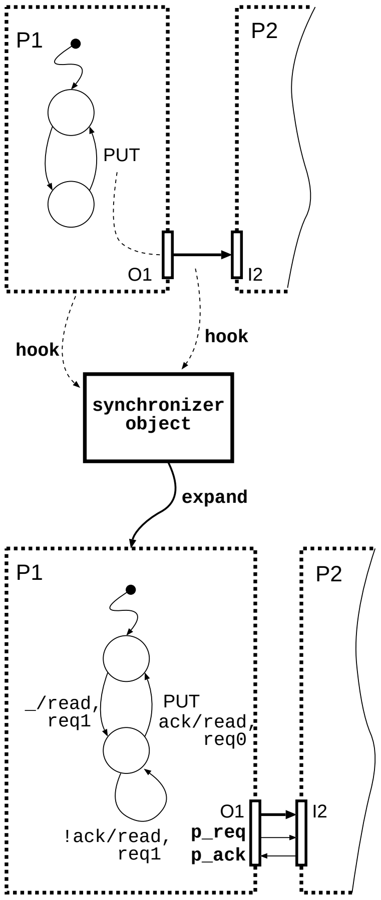


Figure 6: `synchronizer` in action

chronous handshake protocol [11]. The signaling is done through newly created bus objects `p_req` and `p_ack`. The inserted instructions include: `req1` and `req0`, which assert/ deassert the request for data communication, and `read`, which samples the acknowledge bus. The sampled value is used as a transition condition in the expanded FSM.

The protocol implementation of GET (as for instance in processor `P2`) proceeds by a similar, symmetrical expansion. The parametric expansion algorithm, done by the reuse object `synchronizer` is described in pseudocode as follows.

```
 1: attach_synchronizer(fsm F, bus B) {
 2:   for each transition T in F
 3:     S = source_state(T)
 4:     T.attach_instruction(read)
 5:     if transition.instructions contains B
 6:       create a new transition Tw from S to S
 7:       cond(Tw) = ! ack
 8:       cond(T)  = prev_cond(T) & ack
 9:        for each transition U in F
10:          if target_state(U) = S
11:            U.attach_instruction(req1)
12:          else
13:            U.attach_instruction(req0)
14: }
```

The algorithm shown has still certain limitations. For example, two I/O accesses subject to synchronization in the same transition are not allowed. However, by formulating the synchronization problem as a behavioral reuse problem, the synchronizer object can be readily replaced by a new, more sophisticated one without additional modifications to the original behavior of `P1`.

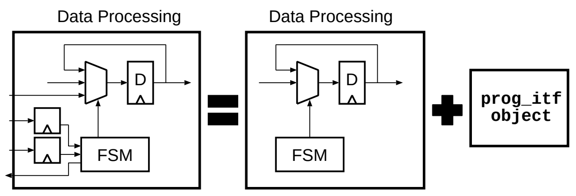


Figure 7: Concept of programming interface object

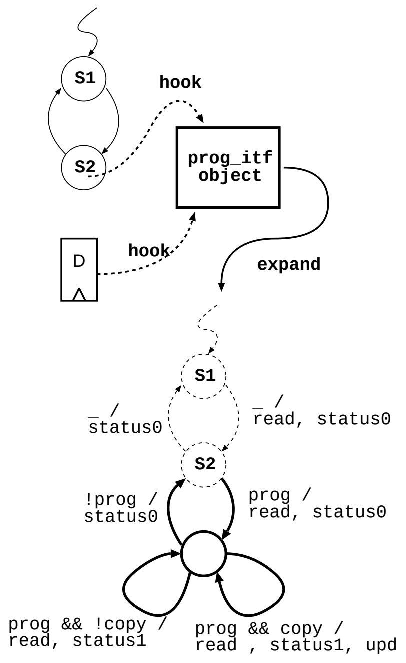


Figure 8: `prog_itf` in action

5.2 The programming interface

The programming interface problem is another natural candidate for reuse at behavioral level. Figure 7 shows the decomposition of the data processing block. The designer is responsible for the description of the data processing itself, but does not need to worry about the protocol with the master interface. Rather, this protocol is available through a reuse object `prog_itf`.

To implement the programming interface in the data processing block, a number of hooks must be given to the programming interface. These include: a reference to the data register for implementing a write operation from the master interface and a reference to a state at which the block can go into programming mode. Given these hooks, the `expand` method of `prog_itf` can be called to implement the programming interface into the block.

An example of the operation of `prog_itf` is shown in figure 8. Data register `D`, as well as state `S2` of the original FSM where hooked onto the programming interface object. Calling the `expand()` method of `prog_itf` modifies the original state transition diagram resulting in one as shown on the bottom of the figure. One new state is inserted, as well as four new transitions. In addition, four new instructions

	Line Count				
	Reuse	Body	Headers	System	Total
CABLE OO RT-C++	1746	5369	1975	4023	13113
CABLE RT-VHDL		21798	5654	2180	29631
DECT OO RT-C++	800	8776	2286	1192	13054
DECT RT-VHDL		19781	6271	2311	28363

Table 2: RT line counts for 2 example designs

are inserted, needed for writing into the `D` register (`upd`), signaling the block status (`status0`,`status1`), and reading the master interface commands `read`. This last instruction also requires the creation of two new condition registers (`prog` and `copy`) to hold these commands.

It is seen that the programming interface is an ideal candidate for reuse, since it is independent of the behavior in which it is embedded.

5.3 Application results

Finally we summarize the results obtained by applying this method to two 80 Kgate designs. In a cable modem demonstrator that we developed recently, an I2C programming interface was designed as a behavioral reuse object. This object was applied to 6 different data processors in the modem. The second design is a DECT base station transceiver. This modem was a first of a kind device and needed an in-circuit debugging interface with real-time observation of registers. Also this debugging interface was designed as a behavioral reuse object. Both of the reuse objects (I2C and debugging interface) affect datapath and control descriptions at the same time, and are similar to the programming interface object (section 5.2).

Table 2 indicates the source code statistics for the upstream Cable Modem and the DECT base station transceiver. For both designs, the first line indicates the C++ line count in the OO-RT model. The RT-VHDL line count of generated code is shown on the second line. The type of code is subdivided in

- `Reuse`: Reusable objects such as programming interfaces according to the presented methodology.
- `Body`: Line count of individual block bodies.
- `Headers`: `.h` files for C++ and entity declarations for VHDL.
- `System`: The system level netlist and testbench drivers.

The savings in coding become obvious from consideration of the total line count. In VHDL, the reused objects get instantiated in the body of blocks, which increase considerably.

6 Conclusions

In this contribution, we have presented a method for behavioral reuse. The difference with current, structural reuse, is that the reuse interface is defined at the behavioral RT level. The RT descriptions are entered in an object oriented environment. The following are essential advantages of this reuse method:

- Reuse is encouraged and supported as an integral part of the design process itself. This is different from the traditional view on reuse, which revolves around the matching and glueing of existing, incompatible blocks.
- Dislike functions in the same component can be developed independently.
- Reusing functionality instead of structure enables compact descriptions that are more easy to understand and maintain.
- Distribution of the reusable objects can be done as object code. Therefore, intellectual property of a reused function is safeguarded.

The method has been applied with success to the design of an upstream cable modem and a DECT transceiver which have been brought to working silicon.

Acknowledgments

This work was carried out under the Flemish Impulse Program for Information Technology (IT-ISIS).

References

[1] P. Ashenden, P. Wilsey, and D. Martin. Reuse through genericity in suave. In *Proc. VIUF 1997 Fall Conf.*, pages 170–177.

[2] B. Djafri and J. Benzakki. Oovhdl: Object oriented vhdl. In *Proc. VIUF 1997 Fall Conf.*, pages 54–59.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oeriented Software.* Addison-Wesley, Reading, MA, 1994.

[4] R. K. Gupta and S. Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, pages 72 – 80, April-June 1997.

[5] Ocapi Homepage. http://www.imec.be/ocapi.

[6] G. Lehmann, B. Wunder, and K. Muller-Glaser. A vhdl reuse workbench. In *Proc. EDAC 1996*, pages 412–417.

[7] G. Martin. Design methodologies for system level ip. In *Proc. DATE 1998*, pages 286–302.

[8] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens. A programming environment for the design of complex high speed asics. In *Proceedings 35th Design Automation Conference*, pages 315 – 320, San Francisco, CA, 1998.

[9] C. Schneider and W. Ecker. Stepwise refinement of behavioral vhdl specifications by separation of synchronization and functionality. In *Proc. EURODAC 1996*, pages 509–514.

[10] G. Schumacher, W. Nebel, and C. von Ossietzky. Object-oriented modeling of parallel hardware systems. In *Proc. DATE 1998*, pages 234–241.

[11] S. Vercauteren and Bill Lin. Hardware/software Communication and System Integration for Embedded Architectures. *Design Automation of Embedded Systems, Kluwer Academic Publishers*, 2:1–24, 1997.

[12] C. Weiler, U. Kebschull, and W. Rosenstiel. C++ base classes for specification, simulation and partitioning of a hardware/software system. In *Proc. ASP-DAC 1995, CHDL 1995, VLSI 1995*, pages 777–784.