Microelectronic System-on-Chip Modeling using Objects and their Relationships*

Frederic Doucet, Rajesh K. Gupta
{doucet, rgupta}@ics.uci.edu
Center for Embedded Computer Systems
University of California
Irvine, CA 92697-3425 USA

Abstract

System-on-chip design introduced many new challenges for engineers. Among them, complexity management and reuse are important issues. Many efforts address complexity by raising the level of abstraction to increasingly functional levels. However, at any level of abstraction, structural information is as important as functionality for microelectronic design. In this paper, we provide an overview of system level design and we present a system engineering methodology that is built upon the definition of objects and their relationships, and their implementations in the context of hardware design. We show how class interfaces can be used for structural representation throughout all design abstraction levels. The object-oriented methodology allows intellectual property (IP) reuse through object libraries and design patterns, and automatic documentation generation. Our approach allows a designer to build a large scale executable, simulatable as well as syntesizable system model using the same language platform. The design process employs an extended UML (Unified Modeling Language) notation, sets of classes and patterns for hardware semantics, and uses SystemC or Cynapps for simulation. We describe the implementation of prototypes that supports our design methodology.

1 Introduction

In recent years, there has been tremendous growth in silicon integration capacity. It is now possible to have a whole system, such as a complete computer, integrated onto a single chip. However, the increase in circuit engineering productivity has not followed the integration density increase at the same pace. This is known as a productivity gap between technology and CAD tools.

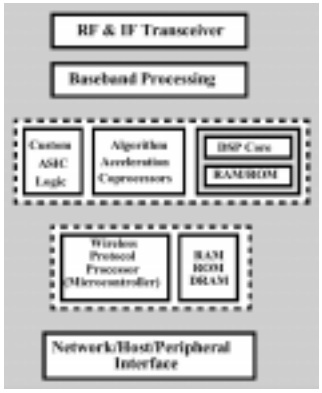

Figure 1 shows a composition for a system-on-chip (SOC). It usually consists of following blocks: processor, memory and caches, wireless system interfaces, network interfaces, sensor and actuators integrated on the same silicon die. SOC designs also include application software and runtime systems. The design is termed system level because it consists of components designed separately and independently. A SOC is a system level design for microelectronic implementation.

Figure 1 A block diagram example of a System-on-chip

* This work was supported by the NSF under grant CCR98-06898, UC MICRO 98-055 and by DARPA-DABT63-98C-0045.

There are several new challenges to SOC designs:

1. Gap between modeling and implementation: system models are written in C, but system implementations are in hardware description languages (HDLs) such as Verilog or VHDL. This leads to a gap since same/similar functionality needs to be re-written from scratch for implementation
2. System simulations: it is hard to carry out complete system simulation because of the diversity of components and the poor simulation efficiency. System level performance parameters such as bit error rate and cache hits ratio are difficult to quantify.
3. Complexity: how will designers handle 20+ million gate designs with current tools?
4. Composability: how will engineers design small blocks that can be integrated seamlessly to a bigger design?

Programming languages have been proposed [1][3][4][8] to address these problems by raising the abstraction to the functional level. Recent years have seen a surge of interest in the use of C/C++ and Java for integrated system modeling as evidenced by the growing use of SystemC [20], Cynapps [21], CoWare [18], OCAPI [17] and SpecC [22] language/libraries. Work on synthesis aspects has been published in [2][6][9]. However, the question remains, how an integrated circuit designer can realize the productivity gains claimed by high-level system modeling languages while ensuring efficient circuit realizations and/or ensuring circuit synthesizability using present high-level/logic-level synthesis technologies?

In this work, we focus on composability. This paper is divided into three parts: (a) fundamental hardware systems modeling needs; (b) objects and relationships that specifically address these needs; (c) a design methodology to support system conceptualization and modeling, design examples and reusable design patterns.

2 System Modeling

Fundamentals of system level modeling are the identification of the model of computation, the language to capture it, and the definition of the architecture implementing the behavior.

2.1 Models of Computation

To compose a design from smaller ones, we need to understand the different models of computation of each component, and how they interact in the global model. A model of computation (MoC) refers to an abstraction of a design functionality. It provides a conceptually higher level view of a design. A MoC is a formal way to articulate a functionality using a set of predefined semantics. A semantic is used to identify an abstract property of a model, for example the notion of a state. MoC formalism and implementations are discussed in [7] and [15].

Good designers know what MoCs are useful to implement a functionality. Different models of computation are often used in different application domains. Common ones are finite state machines, discrete event models, process network, and data flow based variants and equation models. Specifying a design in a formalized model of computation greatly simplifies the synthesis task of any language. Less transformations are needed, so the result is more predictable, and simpler to implement for CAD tools.

2.2 HDL Semantics Necessities

In order to capture microelectronic systems models, we need the following set of semantics available in the hardware description language:

1. Abstraction, to compose large systems by using small ones;
2. Reactivity, to model non terminating interactions with other components, and exception and interruption behaviors;
3. Determinism, for predictable system behavior for a given set of inputs and
4. Simultaneity, for parallelism and multiple clocks.

These semantics were discussed in details in earlier works [4][24], and will not be considered further here. Instead, we focus on language support for SOC modeling.

2.3 Language Issues

A language must provide semantic support to conveniently and efficiently capture specific models of computation. Also, the specification should be executable for fast evaluation of tradeoffs. Many researchers and companies propose and advocate the usage of C, C++ or Java for system level design. Most of them extend the semantic of the language with concurrency, parallelism, and reactivity. Others are proposing complex algorithms for parallelism extraction and other transformations [19]. However, how should a designer formally capture a model of computation using these languages? To our knowledge, only OCAPI [17] supports this feature.

Also, the context in which the specification was written is important. For example, if a company wants to synthesize legacy code into a hardware circuit, it will have to perform successive refinements to transform the sequential call-hierarchy model to a parallel reactive one. On the other hand, if the specification is written from scratch, the designer can explicitly specify these characteristics and have much better synthesis results.

2.4 Platform based Design

System level design methodologies have become an important research topic in the last decade. Many new approaches have been proposed each addressing a particular part of the problem. A major new trend is platform based design [11][12][16].

An application is first described functionally, at the highest level of abstraction, and then mapped to a target platform architecture. The performance is evaluated and iterative refinements are performed to improve the system architecture to satisfy requirements. For example, processes will be ported from software to hardware if they do not meet timing constraints. Another possibility is to widen a processor data path by adding more functional units to handle high data throughput applications. A target platform can be optimized for each application domain while remaining programmable. Reusing platforms that are domain specific and redesigning the software reduces the architectural tradeoffs and shortens time to market.

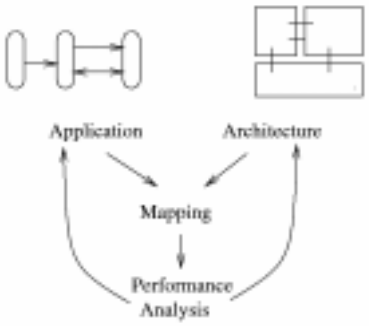

Figure 2 Platform methodology: application-component mapping

In platform based approaches, hardware architectural tradeoffs are driven by software performance requirements: when a software process does not meet them, it is partitioned to dedicated hardware. For the hardware specification, the finest granularity of the architectural exploration is the bus functional level of abstraction. This methodology should be excellent for embedded software design, starting from formal models of computation, translated to elegant software implementations. This will reduce the number of bugs from the average software programmer who does not necessarily understand the model of computation they are trying to program and its underlying semantics in the language. However, the design of SOCs that are processors or large ASICs is not supported appropriately by this methodology because architectural exploration inside hardware blocks needs to be performed.

2.5 Problems with the *Pure Functional* Specifications

System level design using platforms focuses on the functional modeling, independent of the architectural modeling. However, at any level of abstraction, hardware designs always contain a certain amount of structural information that simply cannot be captured at a functional level.

Semantics such as concurrency, reactivity and timing, are sufficient for hardware behavior modeling, but they are not enough to capture the structural information and perform architectural exploration for a SOC design.

For example, a functional model for a processor is its instruction set architecture. However, characteristics such as instruction timing and the number of instructions executed per second, depend on the structure of the implementation: the number of pipeline stages and functional units will change the specification. Other interesting structural elements to model are physical attributes such as size and location of layout blocks and wires, and also power related characteristics .All these structural requirements cannot be captured in a pure functional description.

3 Composability and Sstructure through Objects and their Relationships

In the last section, we identified the need to capture structural information at the highest levels of abstraction and to compose very large designs with smaller blocks. In this section, we will look at the usage of object oriented mechanisms to compose structural and functional elements together.

3.1 The Object Oriented Paradigm

One of the main object oriented design philosophies is to map the physical domain of a problem to a conceptual object structure. The hardware structure can be mapped directly to an object structure with a one-to-one correspondence between objects of both domains.
An objet oriented hardware design has two views:

1. Class diagrams, where behavior and data are encapsulated and structured into classes. They have relationships with each other in order to implement the system functionality through interactions. Class models are built for the design, and for the design attributes.
2. Object diagrams, which is the concrete system view. Objects are instantiated from classes to implement system entities.

The notation we used is an extended UML [14], a specification and documentation notation for object oriented software systems. A class box shows how a class is described. The upper box contains the class name. The middle box contains the attributes, which are the variables inside the class and their types. The lower box contains the methods, which are functions that operate in the scope of the class.

3.2 An Example of Composition: A FSMD

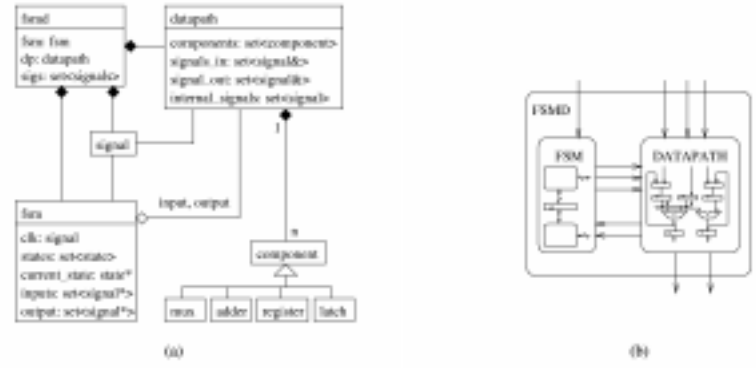

Figure 3 Objects and Relations, FSMD Example: (a) Class Diagram (b) Object Diagram

Abstract relationships between classes can have many meanings for a concrete implementation. Let us illustrate this by using the example of a finite state machine with a data path. Figure 3(a) shows the class diagram, while Figure 3(b) shows the object diagram. In this class framework, we have four main classes: *fsmd*, *fsm*, *datapath* and *component*. There are also four relationships to note:

1. The *fsmd* class is composed of the *fsm* and *datapath* classes. This relationship is noted by a line with a filled diamond on the owner end, in this case on the *fsmd* side. This composition relationship expresses the hierarchy by which the *fsmd* class controls the invocation and liveliness of its composee classes.
2. The *fsm* class aggregates a *datapath* class. The notation is a line with a non-filled diamond on the side of the aggregator. It is an ownership relationship, but much softer than composition. The finite state machine class logically owns the data path, but not structurally. We can see that subtlety in the object diagram, where the structural ownership is present between the *FSMD* and the two *FSM* and *DATAPATH* objects which physically contain them. However, it is not present between the *FSM* and the *DATAPATH* objects. This means that the data path is controlled by the finite state machine functionally, but composed by the *FSMD* structurally.
3. Class inheritance from the component class to the *mux*, *adder*, *register* and *latch* subclasses. Inheritance is noted by a triangle on the base class end of the line. The component class defines an interface for all sub-components. The data path will compose the data path of components using the *component* class interface.
4. Association: *signal* classes provides a communication medium between the components. The *fsm* and *datapath* classes use signals to communicate, but they do not own them. Therefore, it is an association which is specified by a line with no symbols on its ends.

4 Design Modeling with Objects

4.1 Design Patterns

Patterns are extensible known solutions to reoccurring design problems [10]. For example, the decomposition of a certain model of computation into a known hardware structure is a design pattern (for example, the structure of a generic FSMD implementation). Patterns are useful to capture intellectual property for reusability. We need to define design patterns to manipulate objects at every appropriate level of abstraction, in the system level design space. For SOC design, there is a need for two categories of patterns:

1. Models of computation;
2. Component specification, manipulation and binding.

With application programming interfaces (APIs) to manipulate these abstract semantics, we can specify hooks for generic tools and provide a framework based on object patterns. Such APIs and frameworks promote interoperability between tools and formats.

4.2 The Design Framework

Figure 4 shows the class diagram for the composition of a design in a design framwork. A design aggregates a number *n* of design units. A design unit aggregates a model of computation, and optionally, a set of components. A *model_of_computation* is a base class interface: finite state machine, process network, communicating sequential processes and data flow models are derived from it. The *component* class is also a base class. All derived components, whatever the level of abstraction, have this interface for manipulation.

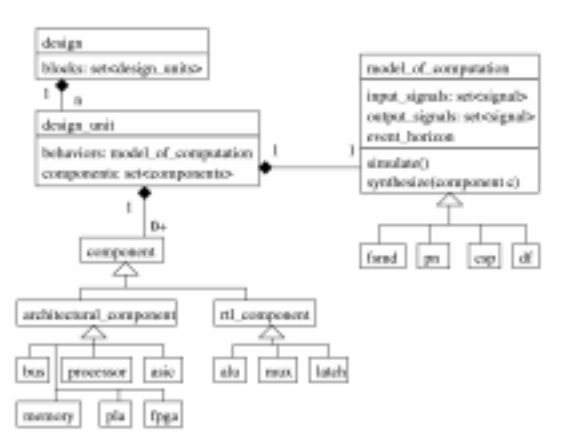

Figure 4 Design Framework Class Diagram

A design unit can be refined, as the model of computation is decomposed into smaller units. Components can be allocated and implicitly binded to the model of computation. This work is not yet complete; there should be composition rules that supervise the refinement as objects are decomposed and the granularity of the specification is refined. Not all models of computation can be associated with any component, and there are rules to be applied to guide the decomposition of models of computation.

4.3 Physical Layer Framework

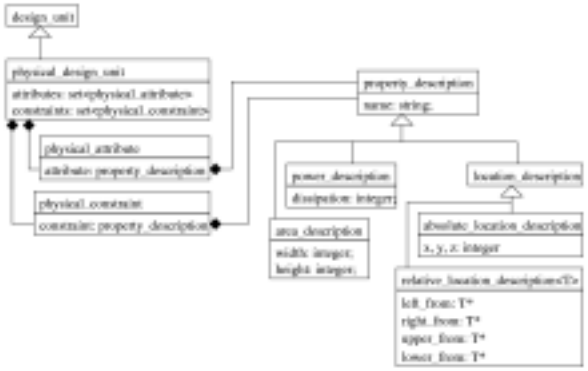

Figure 5 Physical Properties Framework Class Diagram

Figure 5 presents the framework to annotate physical information to a design. Physical predictability in the implementation is captured as sets of annotations to a design unit. We specialize the *design_unit* class into a *physical_design_unit* subclass that can be annotated. Since it is a derived class, we can substitute design units by physical design units as the design is refined. Physical attributes and constraints are aggregated in the *physical_design_unit* class. They take their significance from their composition of a *property_description* class interface. For example, a *physical_attribute* for the location of a design unit on the floor plan will be composed of a *location_description*.

Properties should be specifiable at any level of abstraction for estimation. For example, delays could be specified at the architectural level to do exploration. After technology binding, they can be back annotated to the architectural level for fast simulation with the exact values. In this framework, physical properties are area, location, and power. This design is made to be extensible by making new *property_description* specialization.

4.4 Design Examples

This subsection will present two design patterns. While it is not possible to comprehensively cover all useful patterns, these two illustrate well our modeling paradigm.

4.4.1 Bus-protocol Pattern

Figure 6 On Chip Bus Pattern: (a) Class Diagram (b) Object Diagram

Figure 6 shows a bus-protocol design pattern. It is used to specify on chip bus structures and associated protocol behaviors. Figure 6(a) shows the class diagram, Figure 6(b) the object diagram. A computation aggregates a protocol class, which is a subclass of the *channel_interface* base class. Calls to *read()* and *write()* functions performs data transfers with the appropriate handshakes. Polymorphism is used to encapsulate behaviors behind interfaces.

This design does not contain physical information. Its primary intention is to separate communication from computation [7], and to be able to change the protocol or the bus without affecting the computation behavior to tackle IP interfacing problems. We programmed this design in C++, and used Cynlib for simulation.

4.4.2 DLX Processor Pipeline

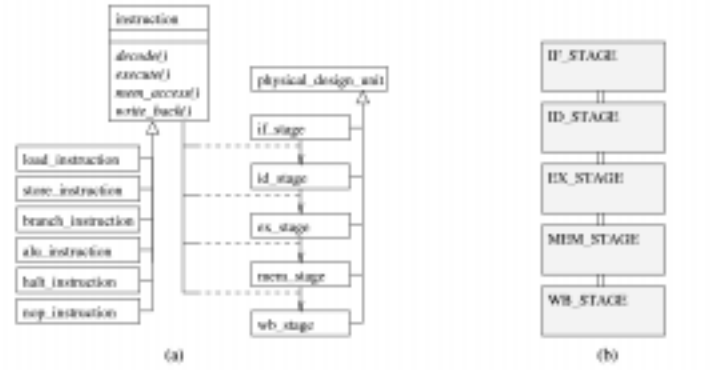

Figure 7 DLX Processor Architecture Pattern: (a) Class Diagram (b) Object Diagram with physical information

As another example, Figure 7 shows a design pattern for a DLX processor pipeline. Figure 7(a) shows the class diagram, and Figure 7(b) shows the object diagram. The pipeline is composed of five stages, each derived from the *physical_design_unit* class. A reactive model of computation is used to express their functionality. Relative location physical attributes are used to specify the location of each stage on the floor plan in relation to each other.

We used the inheritance relationship to specialize instructions from the base class named *instruction*. Polymorphism is used to do the behavior mapping on the pipeline structure. The

base class has a set of methods, and each stage invokes an instruction method to implement its behavior. For example, the *EX_STAGE* stage calls the *execute* method whose call is propagated to the appropriate sub-class. The dashed lines between the lines between the stages and the *instruction* base class means that the *instruction* class characterizes the association between them. In other words, the content of the communication is an instruction subclass.

This design is extendable to add more instructions and also to add or remove pipeline stages. Encapsulation makes the changes to extend the model very modular for this design. We programmed this design in C++, and used SystemC for the simulation. The code consists of 2000 lines of C++, 1800 to specify the structure, and 200 implement the functionality through the instruction functions.

4.5 The Methodology

We now describe a methodology to use object models for microelectronic design. The software engineering object oriented design methodology [13] is appropriate for behavior exploration, but we need to incorporate the structural hardware elements into it. The methodology consist of four steps:

1. Clearly identify the model of computation and write an early model specification with it and use a specific semantics to capture the MoC formally;
2. Once the functionality is expressed satisfactorily, the next step is to identify how many design units to have in the system. This is the model of the target architecture;
3. Distribute the functionality to the architecture by binding models of computation to design units. As demonstrated, we can also use the object oriented model of computation to use polymorphism to distribute the functionality *across* the architecture;
4. Perform iterative refinement to compose the design with smaller design units. Models of computation are decomposed into sub-models and rebinding them to a growing number of components performs architectural exploration and synthesis.

The goal of this approach is to reach a model of computation-component binding with a fine enough granularity for implementation. Automation becomes useful at some levels of abstraction. For example, we can use the Design Compiler to decompose and bind finite state machines with data path to RTL components.

4.6 The Tools

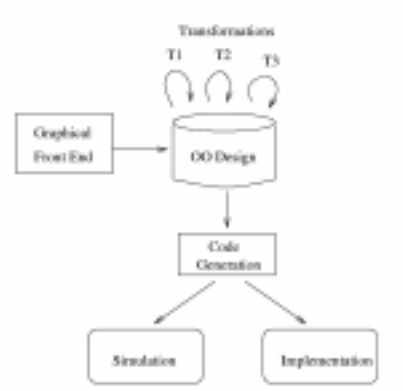

Figure 8 the plan for the tools

In order to support our methodology, we are working on a tool with a graphical user interface based on UML/OMT notations. Abstract and concrete views of the system are to be captured through class and object diagrams described earlier.

The design should be stored in an object database, using abstract semantics. To generate code for either simulation or synthesis, we shall use languages that cover the full set of needed semantics. For simulation, we can use diverse libraries including those by Cynapps and SystemC. The specific choice of the library is not an issue as long as required semantics can be expressed.

5 Summary and Conclusion

In this paper, we provided an overview of system level design. New languages and methodologies were discussed, and their ability to capture large component designs such as processors was examined.

Also discussed was the importance of structure at the highest levels of abstraction in the specification, and how to address this issue by using object and relationships modeling. The topics covered included model semantics, design pattern reusability, and an implementation methodology. However, there still remain numerous issues to resolve:

- How should object oriented mechanisms be synthesized when they are not properly used in the models of computation?
- How do we formalize the object-oriented model of computation?
- How many class frameworks are needed? For which levels of abstraction?
- How should design pattern libraries be extended and maintained?

In conclusion, the authors of this paper believe than object oriented modeling for hardware design is a natural step in the right direction, and a first one to move above the RTL level of abstraction towards the system level design space.

References

[1] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf (June 1994) *Object-oriented techniques in hardware design*, in IEEE Computer

[2] W. Wolf (July 1996) *Object oriented cosynthesis of distributed embeddded systems*, in ACM TODAES

[3] C. Weiler, U Kebschull and W. Rosenstiel (1995) *C++ base classes for specification, simulation and partitioning of a hardware/software system* in VLSI95

[4] R. Gupta and S. Liao (April-June 1997) *Using a programming language for digital system design*, in IEEE D&T of Computer

[5] J. A. Rowson and A. Sangiovanni-Vincentelli (1997) *Interface-based design*, in DAC

[6] L. Semeria and G. Demichelli (1998) *Spc: synthesis of pointers in c*, in ICCAD

[7] E. Lee and A. Sangiovanni-Vincentelli, (Dec 1998) *A framework for comparing models of computation*, In IEEE TCAD

[8] T. Kuhn, W. Rosenstiel and U. Kebschull (1999) *Description and simulation of hardware/software systems with java*, in DAC

[9] Radetzki, M.; Nebel, W. (1999) *Synthesizing hardware from object-oriented descriptions in* FDL

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides, (1995) *Design patterns: elements of reusable object-oriented software*, Addison Wesley

[11] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly and L. Todd (1999) *Surviving the soc revolution: a guide to platform based design*, Kluwer Academic Publishers

[12] D. Gajski, J Zhu, R. Domer, A. Gerstlauer and S. Zhao (2000) *Specc: specification language and methodology*, Kluwer Academic Publishers

[13] R. Pressman (1997) *Software engineering a practitioner's approach, 4th edition*, McGraw Hill

[14] B. Powell (1999) *Real-time uml 2nd edition: efficient objects for embedded systems*, Addison-Wesley

[15] Ptolemy system home page: http://ptolemy.eecs.berkeley.edu

[16] Cadence VCC home page: http://www.cadence.com/eda_solutions/hcd_l3_index.html

[17] OCAPI home page: http://www.imec.be/ocapi

[18] CoWare N2C home page: http://www.coware/com

[19] C Level Design home page: http://www.cleveldesign.com

[20] SystemC home page: http://www.systemc.org

[21] Cynapps home page: http://www.cynapps.com

[22] SpecC home page: http://www.cecs.uci.edu/~specc

[23] VSIA System level design workgroup: http://www.vsi.org

[24] Gigascale Silicon Research Center home page: http://www.gigascale.org