YAML: A Tool for Hardware Design Visualization and Capture

Vivek Sinha[*], Frederic Doucet[*], Chuck Siska[*], Rajesh Gupta[*], Stan Liao[†], Abhijit Ghosh[†]

[*]Center for Embedded Computer Systems, University of California, Irvine
[†]Advanced Technology Group, Synopsys Inc. Mountain View

Abstract

Design visualization is an important part of the system design process. In practice, systems are often visualized using a combination of structural and functional entities. In this paper, we describe an approach that helps to capture the structural aspects of a design at a high level of abstraction and enables the system designer to enter designs "schematically" using predefined structural and functional entities conforming to UML notation. The corresponding tool, YAML (Yet Another UML front end) provides support for modeling objects and a range of object relationships that are crucial to real-life embedded system designs. A YAML design entry can then be automatically translated into synthesizable C++ code for simulation and hardware synthesis.

1 Introduction

Increased complexity of microelectronic systems demands an increase in the level of abstraction at which these systems are modeled. Current efforts in this direction seem to have been focussed on use of high-level programming languages and associated software process engineering tools in building complex systems on chip. Such efforts include object oriented modeling paradigms [18], and more specific approaches using C/C++/Java as a hardware modeling language [17, 16, 10, 9]. While such efforts enhance the descriptive ability by raising design descriptions to increasingly functional levels, they ignore a hard fact: that at *any* level of design description, microelectronic systems always contain certain amount of structural information that simply cannot be captured at a functional level. More importantly, this information is often critical for early design/architectural decisions. In this paper, we examine the nature of structural information and present a method of incorporating such information even in the most functional C++ descriptions. We also explore the usage of UML in object oriented hardware modeling that enables a designer to visualize microelectronic system designs.

The Unified Modeling Language (UML) [8, 3] is a language for specifying, visualizing, constructing, and documenting the artifacts of systems. It provides a convenient notation that allows the developer to capture structural and behavioral details using an object oriented design methodology. Its most prevalent use has been in the design and documentation of large scale software systems. Recently, UML has been explored for its application in hardware modeling [11]. This is particularly relevant in view of increasing use of object oriented languages for hardware modeling, like C++ or Java. Projects like SystemC [13, 15], Cynlib [5] and OCAPI [16] use class libraries to support hardware modeling. SpecC [9] uses a superset of C at the specification level, but generates an intermediate C++ model of the program, and uses C++ classes to model behavior and communication in the system. Writing hardware descriptions in C++ using class libraries, can be quite tedious. Design visualization tools are needed to help a designer conceptualize, model and refine system design without being burdened by the implementation and syntactic details of the C++ class libraries.

Towards this end we have built a design entry tool YAML, which uses UML notations to model hardware, and allows the user to input information about objects and relationships into a UML class diagram (for behavioral hierarchy) and an object diagram (for structural modeling). This is in contrast to the traditional use of UML, primarily as a documentation language. YAML generates the C++ code for the design, using the information input by the user to the UML class and object diagrams. Also, the object diagram can be used later during hardware synthesis to provide detailed structural information. We note that the embedded system modeling capabilities are not defined by UML, which has limited support for specifying concurrency, timing and placement. In fact, we extend the UML notations to provide support for these hardware specific features [7].

We briefly describe SystemC class library, and how we extend it to model structural information in Section 2. We describe our approach to design visualization using YAML in Section 3, followed by a description of our implementation and conclusions.

2 System Modeling using SystemC and ICSP

As system complexity increases and design time shrinks, it becomes important that system specification be captured in a form that leads to unambiguous interpretation by the system implementers. The usage of object oriented languages for hardware modeling has provided a good alternative [12, 16]. This has enabled many system, hardware, and software designers to create executable specifications for their systems, which are for the most part, functional models written in a language like C or C++. These languages provide the control and data abstractions necessary to develop compact and efficient system descriptions. However, a programming language like C or C++ is intended for software and does not have the constructs necessary to model timing, concurrency, and reactive behavior, all of which are needed to create accurate models of systems containing hardware [10]. An object oriented programming language like C++ provides the ability to extend the language through classes [17], without adding new syntactic constructs. A class-based approach to providing modeling constructs is superior to a proprietary new language because it allows designers to continue to use the language and tools they are familiar with. These classes add to the C++ language, the capabilities needed to model hardware.

2.1 SystemC

The fundamental building block in the SystemC class library [15] is a process. A process is like a C or C++ function that implements behavior. A complete system description consists of multiple concurrent processes which can be synchronous or asynchronous. Processes communicate with one another through signals, and explicit clocks are used to order events and synchronize processes. In addition a module, which is a container class can be used to model hierarchy.

Using the SystemC library, the user can model a system at various levels of abstraction. At the highest level, only the functionality of the system may be modeled. For hardware implementation, models can be written either in a functional style or in a RTL (register-transfer level) style. The software part of a system can be naturally described in C++. Interfaces between software and hardware and between hardware blocks can be described either at the transaction-accurate level or at the cycle-accurate level. Moreover, different parts of the system can be modeled at different levels of abstraction and these models can co-exist during system simulation. C++ and the SystemC classes can be used not only for the development of the system, but also for the testbench. SystemC consists of a set of header files describing the classes and a link library that contains the simulation kernel. Any ANSI C++ compliant compiler can compile SystemC, together with the program. During linking, the SystemC library, which contains the simulation kernel is used. The resulting executable serves as a simulator for the system described.

2.2 Modeling Structure using the ICSP Class Library

At the most abstract level, structure refers to an incidence structure commonly described as "netlists" of various sorts. Often such structures are described using declarative semantics in most hardware description languages where a block consists of a list of components and the order of specification of the component list is not important. However, for high performance microelectronic system designs, incidence structure is not enough. Indeed, relative placement (and floor-planning information) is often needed to make architectural design tradeoffs.

The Incidence Composition Structure Project (ICSP) class library [14] is a set of class interfaces extended from the SystemC class library to express structural layout, routing and floor-planning information. Though SystemC provides the concept of hierarchical containment through the SystemC module class, there is no provision to specify the relative placement of components, and the attachment or placement of ports with respect to the components. The ICSP class library allows the user to express structural information like component and port placement and layout. This information can be very useful during the later stages of hardware synthesis, specifically during placement and layout, and can be captured during the high level specification stage itself. Structural information is an important part of datapath block designs in most ASIC and custom IC designs [4]. The SystemC component interfaces have been specialized to contain the following information in the ICSP class library.

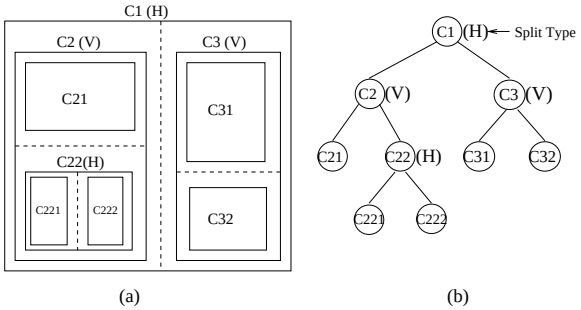

Figure 1. Component order and placement: (a) Rectangle partitioning, (b) Corresponding Slicing Tree

- *Component order & Placement*: Components are represented by rectangles and their relative placement is

specified using a slicing tree consisting of horizontal and vertical composition of rectangles. Figure1 shows an instance of rectangle partitioning and the corresponding slicing tree. A component can be split either horizontally or vertically into subcomponents, and each of those subcomponents can be split further. This is implemented in the *icsp_module* class, which can be used to specify the type of split (horizontal or vertical) and also the order of subcomponents.

- *Ports Locations*: The location of input and output ports can be specified on the sides of a component. For instance in Figure 2, if we consider subcomponent S1, the position of input port *in1* is "left upper" while that of output port *diff* is "right lower".

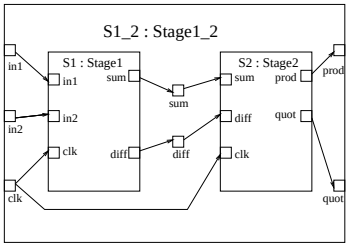

Figure 2. The partitioning of module S1_2 of type Stage1_2 horizontally into two subcomponents S1 and S2. Port locations are also shown.

The goal of the ICSP class library is to reduce wiring congestion, minimize routing overhead and keep blocks with tight timing requirements close to one another. The ICSP class library can be incorporated into a schematic editor for system level designs, where the user can specify the relative placement of components and ports. This is described in the next section.

3 Visual Modeling and Design Entry

3.1 YAML

The motivation behind developing YAML is to help the user to conceptualize and model the design in a graphical interface using extended UML notation. Several graphical editors have been developed for visualization of various UML diagrams. Chief among them is Rhapsody by iLogix [2] and Rational Rose [1]. The primary use of these environments is in design conceptualization and documentation through code generation capabilities to help software development. Along similar lines, YAML has been designed and specifically targeted to include both structural and functional semantics, with the objects and relationships that help design conceptualization and implementation of IC design. This is particularly helpful while using the SystemC and ICSP library of C++ classes, as it allows the user to model the system without the effort required to describe syntactic details of the underlying C++ code.

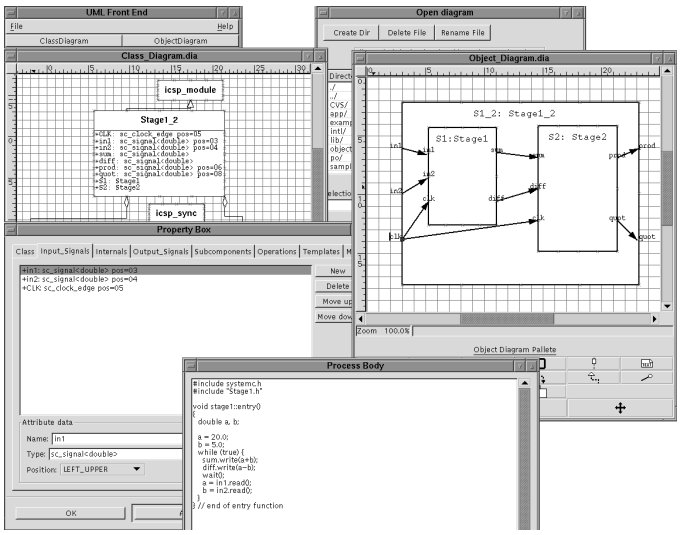

Figure 3. Screenshot of YAML showing the various user interfaces

Figure 3 shows the various user interfaces supported by YAML. It contains class diagram canvas for specifying behavioral hierarchy, and an object diagram canvas for specifying structural details. Each of these canvas has a palette of design entry icons to choose from, and property boxes for each element in the diagram. YAML allows the user to do a schematic like design entry of the system using UML notation.

3.1.1 The Class Diagram

The class diagram (figure 4) in YAML is used to specify the behavioral hierarchy of the system using classes from the SystemC and ICSP class libraries. The user can instantiate a SystemC/ICSP process or module from the class diagram palette, and then add various properties to this class. The input and output signals, and other attributes like internal signals and variables, subcomponents, etc. can be specified in the property box for that class. The process body can be entered into a text window in the property box. Similarly, the body of the constructor can be entered to initialize the internal variables if required. The user can also specify various relationships among classes in the class diagram, as discussed in the next subsection.

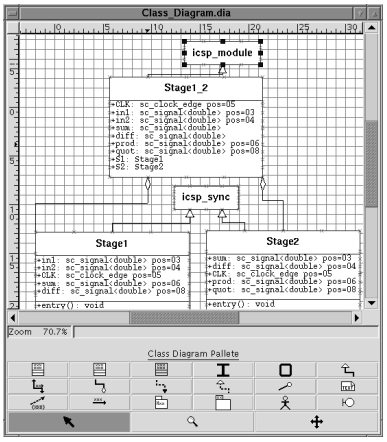

Figure 4. The 2 stage pipeline (Figure 2) modeled in the YAML class diagram using the ICSP class library

3.1.2 The Object Diagram

The object diagram (figure 5) is used to specify structural details of the system. For the SystemC class library, the object diagram merely shows the instances of classes (specified in the class diagram) as objects, and the interconnection of these objects.

On the other hand, for the ICSP class library the object diagram is used to add all the necessary structural information. It allows the schematic placement of components and input/output ports. The user can specify the type of component partition, either horizontal or vertical, the order of subcomponents, and the location of the input and output ports around the component in the object diagram.

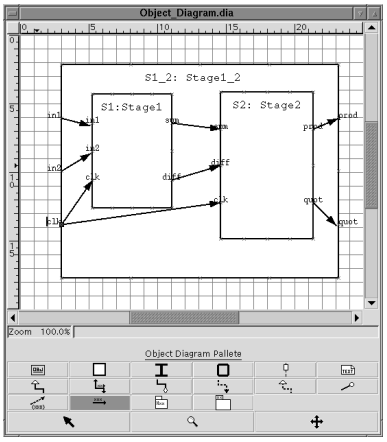

Figure 5. The 2 stage pipeline (Figure 2) modeled in the YAML object diagram

3.2 Objects and relationships

YAML supports system design using objects and relationships. While objects represent function or structure, relationships characterize the interface or interaction among these objects. We have extended the meaning and notation of these objects and relationships so as to support the design of microelectronic systems [7]. The following relationships can be specified in YAML.

1. *Generalization*: Generalization or inheritance is an important concept in object oriented programming. From a base class, we can derive specialized classes, through inheritance. Generalization is primarily used to derive specialized classes from the set of base classes provided in the SystemC and ICSP libraries. For example, in Figure 4 class *Stage1_2* is derived from the base class *icsp_module* using the generalization relationship. It is represented by a unidirectional arrow with a triangular head.

2. *Aggregation/Composition*: An aggregation relationship applies when one object physically or conceptually contains another. Composition is a strong form aggregation in which the owner is explicitly responsible for the creation and destruction of the owned objects. It can be represented by the actual inclusion of the owned object in the owner, or by a directed arrow with a diamond shaped head pointing towards the owner. Composition corresponds to component hierarchy in a design. In Figure 4, classes *Stage1* and *Stage2* are subcomponents of class *Stage1_2*, and are connected to it by diamond shaped arrows pointing towards *Stage1_2* which is the owner.

3. *Association:* Association represents conceptual relationships between classes, and is used for the exchange of messages. It corresponds to communication, either through signals or channels, and is represented by a open directional arrow showing the direction of communication. The association relationship can be extended to contain information like communication protocols between objects or even wire delays at a very low level of design. In Figure 4, class *Stage1* has association relationships with class *Stage2*.

YAML provides support for interpreting the relationships between classes, and uses this information to generate code. For instance, generalization can be used to create a derived class, and YAML uses this information from the UML class diagram to reflect this derivation in the code generated for derived classes.

We have provided handles in the data structure of a class, so as to access its neighboring classes and get any required information while processing it. The neighbors of a class

are all the classes around it in the class diagram that are directly connected to it by some kind of relationship. This can be particularly helpful in interpreting the composition relationship that exists between a module class and the subcomponents declared in it, where we need to access the subcomponents declared in the SystemC or ICSP module. In short, handles provide a mechanism to traverse the graph of connected classes and access information from any of the nodes of this graph as required.

3.3 Code generation from YAML

A major advantage of using YAML is the ability to avoid the complex syntactic details involved in using the C++ class libraries. Writing code is not easy when these class libraries are used. YAML allows the user to perform graphical entry of the system using SystemC and ICSP class libraries, and processes this information to generate the underlying C++ code.

```
#include "Stage1"
#include "Stage2"
struct Stage1_2 : public icsp_module {
/*Internal Signals/Variables*/
   sc_signal<double> sum;
   sc_signal<double> diff;
/*SubComponents*/
   Stage1 S1;
   Stage2 S2;
/*Constructor*/
Stage1_2(const char* NAME, sc_clock_edge& CLK,
   const sc_signal<double>& in1,
   const sc_signal<double>& in2,
   sc_signal<double>& prod,
   sc_signal<double>& quot)
 : icsp_module(NAME,
   ICSP_SPLIT_HORIZONTALLY,
   icsp_corder(icsp_comp(S1),icsp_comp(S2)),
   icsp_porder(icsp_ppair(CLK,icsp_e_lleft),
                icsp_ppair(in1,icsp_e_uleft),
                icsp_ppair(in2,icsp_e_mleft),
                icsp_ppair(prod,icsp_e_uright),
                icsp_ppair(quot,icsp_e_lright))),
   S1("Stage1", CLK, in1, in2, sum, diff),
   S2("Stage2", CLK, sum, diff, prod, quot)
    {end_module();}
};
```

Figure 6. Code generated for the ICSP class Stage1_2

The user can generate the SystemC + ICSP code from YAML, after specifying the various details in the class and object diagrams. Figure 6 shows the code fragment generated for the ICSP class *Stage1_2* from Figure 4. It is to be noted that for the ICSP module class, input and output signals are declared only in the constructor.

In the ICSP code generated above, the component partitioning is specified by ICSP_SPLIT_HORIZONTALLY (horizontal partitioning in this case), component order is specified by the *icsp_corder* function, and port locations by the *icsp_porder* function. The *icsp_ppair* function maps each port to its location around the component. For instance, port "in1" is mapped to the upper left location of component *Stage1_2* (Figure 5).

It can be seen that writing such code directly in a text editor is much more difficult than specifying the details in a graphical design entry tool, and then generating the code automatically.

YAML has been used to model various designs including a DLX compatible processor pipeline [7]. The DLX pipeline code consists of around 2000 line of C++ code most of which was generated automatically by YAML. Only around 200 line of user code containing the SystemC process bodies were entered by the user. Simulation results were produced by compiling and executing the resulting SystemC code. A simple ICSP/SystemC to VHDL translator [6] was used for further synthesis.

4 Conclusions

This paper highlights the importance of structural information at the high level functional description and explains how the ICSP class library helps in specifying detailed structural and layout information at the functional level. Design conceptualization includes not only use of object oriented mechanisms for modeling structural and functional information, but also their visualization. YAML provides a user friendly graphical interface to model systems under the guidelines of UML, using the SystemC and ICSP C++ class libraries. The user can specify the details of SystemC and ICSP classes into the UML front end. YAML frees the designer from the syntactic nuances and details corresponding to these class libraries, so that he may focus on the organization of the structural and functional components of his design. The code generated by YAML conforms to the syntax of ICSP and SystemC classes and can be directly compiled and simulated. YAML is available on Linux as well as Solaris operating systems and is currently under active development.

Acknowledgments

The authors would like to acknowledge UC MICRO 98-055, DARPA DABT63-98C-0045 and NSF CCR98-06898 funds for this work.

References

[1] Rational rose. http://www.rational.com.

[2] Rhapsody. http://www.ilogix.com.

[3] *UML Notation Guide 1.1*. http://www.rational.com/uml, September 1997.

[4] A. Chowdhary and R. Gupta. Extraction of functional regularity in datapath circuits. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18(9), September 1999.

[5] CynApps Inc., http://www.cynlib.com. *Cynlib Reference Manual*.

[6] F. Doucet, S. Kim, and R. Jerjurikar. Icsp/systemc object oriented translation to rtl hdl. Technical Report 07-00, CECS, UC Irvine, 2000.

[7] F. Doucet, V. Sinha, and R. Gupta. System-on-chip modeling using objects and their relationships. Technical Report 99-53, CECS, Univ. of California, Irvine, http://www.ics.uci.edu/∼cecs/, October 1999.

[8] B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison Wesley, 1998.

[9] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[10] R. Gupta and S. Liao. Using a programming language for digital system design. *IEEE Design & Test of Computers*, April-June 1997.

[11] H. Hallal, K. Xiao-Hua, and R. Negulescu. Experiments in modeling integrated circuit blocks by uml. In *International Workshop on IP Based Synthesis and System Design*, 1999.

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

[13] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *34th Design Automation Conference*, 1997.

[14] C. Siska. Icsp technical report. Technical report, CECS, UC Irvine, June 1999.

[15] Synopsys Inc., http://www.systemc.org. *SystemC Reference Manual 1.0*.

[16] S. Vernalde, P. Schaumont, and I. Bolsens. An object oriented programming approach for hardware design. In *IEEE Computer Society Workshop on VLSI*, Orlando, April 1999.

[17] C. Weiler, U. Kebschull, and W. Rosenstiel. C++ base classes for specification, simulation and partitioning of a hardware/software system. In *VLSI 95*, Japan, 1995.

[18] W. Wolf. Object oriented cosynthesis of distributed embedded systems. *ACM TODAES*, July 1996.