Structural Design Composition for C++ Hardware Models

Frederic Doucet, Vivek Sinha, Rajesh Gupta

Center for Embedded Computer Systems
University of California at Irvine

{doucet,vsinha,rgupta}@ics.uci.edu

Abstract

This paper addresses the modeling of layout structure in high level C++ models. Researchers agree that the level of abstraction for integrated circuit design needs to be raised. New languages and methodologies are being proposed, most of them influenced from the software engineering domain. However, one of the fundamental hardware design challenges is often overlooked as push button synthesis solutions are sought: physical design predictability. In this paper we describe how C++ constructs should be used to capture structural and physical implementation concerns. Our explanation relies on the importance of the floorplan and component placement estimations at high levels of abstraction. We highlight how using object oriented mechanisms eases the structural modeling of circuit components, and we present a C++ class library design to specify these structural concerns.

1. Introduction

High level modeling using C/C++ models [11] [2] has been proven efficient for architecture exploration and verification, and the implementability of these models progresses as their productivity gains are demonstrated to engineers and managers.

Although complexity is often cited as the primary design concern, building floorplans and layouts is also a challenge since it is difficult to abstract and to predict in a behavioral specification. A design step such as place and route is both hard and time consuming in deep submicron designs [6]. Timing closure is very difficult when a design goes from logical to geometrical granularity, since most of the weight of timing analysis is now a function of the lengths of the interconnection between gates rather than by the gates themselves.

These are essentially structural concerns, and the new emerging languages do not have the semantics to capture the information needed to do timing based design[3]. For example, wire lengths and block placements are not captured in functional specification where blocks and wires are not yet necessarily identified or allocated.

The connection between C level modeling and physical design is discussed in [8] and in [5]. However, no clear path has yet emerged for the integration of the structural physical information in high level languages. In this paper, we will discuss a semantical approach to specify the structural implementation information in high level models. The implementation of the syntax is based on using an object hierarchy to reflect the physical structure of the implementation as a structural object model. We present the Incidence Composition Structure Project class library to capture layout and floorplanning information in C++ models.

2. Languages and Semantics

Many hardware design methodologies and tools with variants of C/C++ or Java have been proposed in the recent years. The semantical outlines we can clearly distinguish for structural composition in the main languages are the following:

1. VHDL and Verilog: formal structural semantic for process networks and strong connectivity through signals;
2. C: informal semantics, a program is a set of functions with a set of global data structures;
3. C++ and Java: informal semantics, the C functional space augmented with the “object” space where data and functionality is encapsulated.

VHDL, Verilog and their variants are well established for RTL design and have yielded many successful designs over the years. Although their synthesizable semantics are

known and understood, these language are not the best ones for architectural exploration because of their strong connectivity and their lack of functional hierarchy. It is interesting to note that variants of C such as Superlog [12] and SpecC [2] add semantical constructs over the functional design space of C through the usage of keywords. These constructs (such as a *Behavior* in SpecC) have special semantical meaning recognizable by the designers and the tools. In C++, the designer can build such semantical constructs by using classes. The SystemC and Cynlib class libraries are used to give objects process interfaces and to build the design in an environment where a reactive simulation kernel can invoke the design entities.

2.1. Desirable Semantics

A good hardware design language should have the semantics highlighted in [4] available in its syntax: reactivity, concurrency, composability and binary data types. The language choice will have to be made by each company individually, according to the application domain, interoperability concerns and their design process. For more information about semantics, the reader is invited to read the Accelera [9] FSMD for semantical proposal information, and for interoperability the IEEE DATC [10] effort.

2.2. Why Object Orientation Helps

The main object oriented design philosophy is to map the object structure of a problem into an object structure in the software model. Hence, a hardware structure can be mapped directly to an object structure with a one-to-one correspondence between the objects of both domains. This object structure in the model can be analyzed and annotated with information. This is how object models are extendible without disrupting the existing tools.

3. Structural Layout

Structure involves the connectedness of components, the orientation and how they are connected. In a high level design language, a designer must describe several types of structure:

- what components nest inside others – parent-child relationships
- how sub-components are juxtaposed – sibling relationships e.g., the incidence structure in terms of north, south, east, west
- what ports are attached to the component
- where on the component those ports are attached

A netlist is an example of some of these kinds of structure. In addition to juxtaposition relationships such as "part A adjoins, and is to the left of, part B", structure can include more detailed information such as component size and routing channels.

The structure which we consider in this project consists of component hierarchy and juxtaposition. While other aspects of structure can be useful to a designer, we have chosen to begin the exploration of structure-added descriptions with juxtaposition.

The approach to describe the structure is to rely on rectangle partitioning. Each component is considered to have a rectangular outline. The placement description involves the subdivision of a component's rectangle. An example subdivision, or partitioning, description might be the following:

- Part A is decomposed into parts B and C with B to the left of C

Thus, rectangle partitioning description can combine both hierarchy specification and placement. Alternatively, in some circumstances it may be appropriate to ignore placement for the moment. In the above example, such a description might be the following:

- Part A is decomposed into parts B and C

The rectangle partitioning style does not handle fuzzy structural relationships very well. On the other hand, it is pretty easy to visualize and understand. The results of changes are fairly intuitive. Also, it is obvious that such a structural description results in a flat component layout. No planarity testing required.

A partitioning concept such as "to the left of" is both simple to visualize and understand, and it is fairly easy to implement. There are two directions in which to perform a 2-for-1 type of partition, either horizontally or vertically.

The most common such partitioning is a 1-dimensional array partition. It is common to stack, or abut, more than two components in a placement array. Describing this in terms of binary partitions is quite cumbersome. Thus, we consider that being able to describe the partition of a rectangle into any number of horizontal sub-components (or vertical sub-components) is important. This array partitioning allows the clustering of components at the same level where they are conceptually equal – e.g., layout of an array of pipeline stages.

For this work, we chose to represent the hierarchical containment of components by rectangles and their relative placement. The structure is specified using a slicing tree [7] 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. On

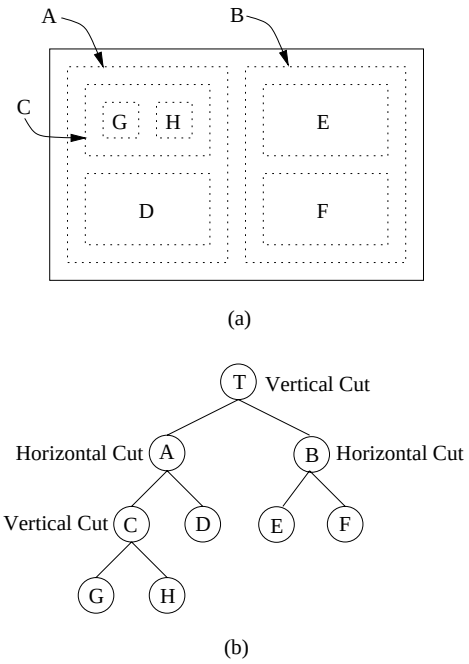

Figure 1. Component order and placement: (a) Rectangle partitioning, (b) Corresponding Slicing Tree

Figure1(a), we see that the top level component T is split horizontally into subcomponents A and B, through a vertical cut. Subcomponent B is split vertically into subcomponents E and F through a horizontal cut. The slicing tree on Figure1(b) shows the top node and the child nodes. Composite nodes (the nodes that are split) are decorated with the cut type, valued horizontal or vertical. The location of input and output ports is specified on the sides of a component, but it does not appear on this tree.

4. ICSP Structure Extensions

Modeling structure in our project, involves describing two different kinds of things: the subdividing of a component into sub-components with relative placement, and the mapping of a component's ports onto locations around the periphery of that component. For the implementation of the classes, we choose to start with the SystemC [11] C++ class interfaces.

While SystemC provides the concept of hierarchical containment, missing are the concepts of relative placement of modules and the attachment or placement of ports with respect to their components. In addition, ICSP introduces component and port classes to support the representation of placement on par with the module class.

The icsp_module class provides the ability to indicate both port locations and the layout of sub-components. An object of class ICSP module is treated as a rectangle. The attachment locations on the edges of the rectangle can be specified for each of its ports. Sub-components can either subdivide the rectangle horizontally or vertically, but not both. To effect subdivision in both directions one more layer of hierarchy is required in the description – for example, to create a 2x2 subdivision a dummy layer of two dummy sub-components could split the rectangle horizontally (i.e., side-by-side) and then the left dummy sub-component could be split vertically into the two actual sub-components in the left 2x2 column, and similarly in the right dummy sub-component for the other two actual sub-components. Note that sub-components are typically specializations of the SystemC module class class. These specializations can include other icsp_module instances or SystemC module instances. We will walk through an example of the composition of a DLX pipeline to illustrate the usage of the ICSP classes.

4.1. Composite Nodes

Only composite nodes of the split tree can be horizontally or vertically split. An ICSP module is a composite node.

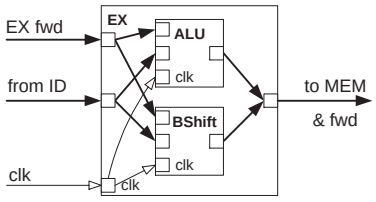

Figure 2. Vertical split and port locations for EX stage

Figure 2 shows the usage of an icsp_module to group sub-modules together with signal placement. This is a grouping of two functional units, themselves ICSP module class. This is the ICSP equivalent of the SystemC module. The module is an EX pipeline stage and the sub-module are two functional units, ALU and BShift, laid out vertically within the module. The three incoming signals are located on the left side low, middle and high areas of the EX module's periphery. These signals are then split and passed into the sub-modules. All classes include signal placement. This figure corresponds to the source code of "struct stage_ex".

The code below shows how the partitioning and port attachments shown in Figure 2 are described in an ICSP class. The syntactic elements beginning "icsp_" refer to ICSP library parts. The functional units are called alu1 and bsh1.

```
struct stage_ex : public icsp_module {

  alu    alu1;   // sub-modules: both EX stage functional
  BShift bsh1;

  icsp_inport< bool >         clk
  icsp_inport< t_id_ex_reg >  id_ex_in;
  icsp_inport< t_ex_mem_reg > ex_mem_in;
  icsp_ouport< t_ex_mem_reg > ex_mem_out;

  // constructor.
  stage_ex( const char * NAME )

      // Initialize base module:
      // 1. rectangle partition
    : icsp_module( name,
          ICSP_SPLIT_VERTICALLY,
          icsp_corder( icsp_comp( alu1 ),
                       icsp_comp( bsh1 ) ),

      // 2. Port placement.
          icsp_porder(
            icsp_ppair( clk,        ICSP_LOWER_LEFT  ),
            icsp_ppair( id_ex_in,   ICSP_UPPER_LEFT  ),
            icsp_ppair( ex_mem_in,  ICSP_MID_LEFT    ),
            icsp_ppair( ex_mem_out, ICSP_UPPER_RIGHT ))
      )
...
```

This stage_ex class is inherited from the icsp_module class, which itself inherits from SystemC sc_module class. The two sub-processes, the ALU and the barrel shifter are defined as internal to this EX stage. Next are four ICSP ports declarations. The rest of the code is a C++ constructor definition for the module. This constructor takes one argument, the instance name of the module. After the module name, the icsp_module class initialization consists of a subdivision of the module vertically followed by both the order of the sub-components within the module.

The alu1 and bsh1 sub-processes are each wrapped in an icsp_comp class which supports a simple daisy-chain linkage, a simple mechanism to allow ICSP to track the order of the horizontal or vertical stack of sub-parts comprising the module. The icsp_corder function performs the component ordering and linking. Sub-parts of an ICSP module can be either SystemC or ICSP flavors. In either case, each is wrapped in an ICSP component class.

The base icsp_module class initialization ends with the specification of port locations: the placement of the incoming and outgoing signals on the periphery of the module. Each port is associated with a module rectangle location. These combinations are called port-location pairs. In this example, the clock port, is placed at the lower left edge, ICSP_LOWER_LEFT, of the stage_ex module rectangle. This locating or placement is performed by the icsp_ppair function. This function takes as arguments a signal or channel of some type and, as a second argument, a location enumeration value. As with the icsp_comp wrapper class, the icsp_ppair wrapper class provides a daisy-chain linkage. The icsp_porder function serves to keep track of the port-location pairs in the ICSP module. The remainder of the code of the constructor is omitted because it is not related to the ICSP features.

4.2. Leaf Nodes

The code fragments defining the sub-module classes, alu and bsu, look similar to this icsp_module with the exception that it is a leaf component in the slicing tree: it does not have sub-components. For example, the alu code, shown below, inherits from icsp_leaf_module

```
struct alu : public icsp_leaf_module {
  icsp_inport< bool >         clk;
  icsp_inport< t_id_ex_reg >  id_ex_in;
  icsp_inport< t_ex_mem_reg > ex_mem_in;
  icsp_ouport< t_ex_mem_reg > ex_mem_out;

  //Constructor
  alu( const char *NAME )
    : icsp_sync(NAME,
        icsp_porder(
          icsp_ppair( clk,        ICSP_LOWER_LEFT  ),
          icsp_ppair( id_ex_in,   ICSP_UPPER_LEFT  ),
          icsp_ppair( ex_mem_in,  ICSP_MID_LEFT    ),
          icsp_ppair( ex_mem_out, ICSP_UPPER_RIGHT ))
     )
...
```

As can be seen, the base icsp_leaf_module class initialization looks similar to the icsp_module class initialization for a composite node. It includes the signal placement via icsp_porder but it is missing the physical sub-division via icsp_corder. This alu module could also inherit from sc_module if there was no port location information.

4.3. Putting it all together

Let us now complete the pipeline by composing it from its submodules. The following code is the implementation of the pipeline composition:

```
struct dlx_pipeline : public icsp_module {

  // sub-modules
  stage_if  IF;
  stage_id  ID;
  stage_ex  EX;
  stage_mem MEM;
  stage_wb  WB;

  icsp_inport<bool> clk;

  // constructor.
  dlx_pipeline( const char * NAME )

    : icsp_module( name,
          // 1. rectangle partition
          ICSP_SPLIT_HORIZONTALLY,
          icsp_corder( icsp_comp( IF ),
                       icsp_comp( ID ) ,
                       icsp_comp( EX ) ,
                       icsp_comp( MEM ) ,
                       icsp_comp( WB ) ),

          // 2. Port placement.
          icsp_porder(
             icsp_ppair( clk, ICSP_LOWER_LEFT  ))
      )
...
```

Since the pipeline is a composite module, it also derives from the ICSP module class. The composed nodes are the ID, IF, EX, MEM and WB stages. The pipeline has

one explicit input, the clock signal. In the constructor, the pipeline is first split horizontally and the components are daisy chained from left to right. Secondly, the clk port is placed in the lower left corner of the outer rectangle. Figure 3 illustrates the geometrical composition and port location information.

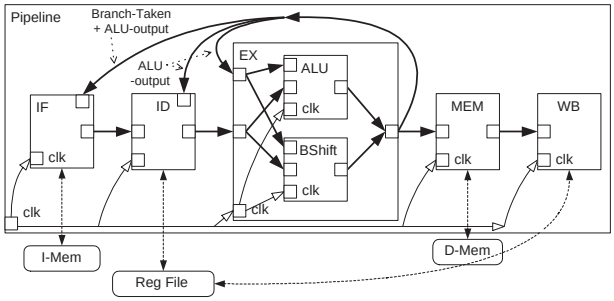

Figure 3. Horizontal split and port locations for DLX pipeline

5. Conclusion

In this paper, we first highlighted the importance of structural information in high level models. Then, we gave an overview of what structural information is capturable in currently available languages. Then, we defined a set of semantics to capture the structural information for layout composition. Finally, we described an implementation of these semantics in the ICSP class library.

This information can be useful on the multiple levels of floorplanning activity: overall chip layout, optimization of individual cores and critical path optimization. The primary use of the ICSP library is in definition of structure to aid the design and synthesis of structural blocks, for data flow or structurally regular datapath designs [1]. Future work is to use the ICSP class interfaces to characterize RTL class libraries to do datapath composition and to add cell size and wire length attributes.

6. Acknowledgments

The authors would like to acknowledge the work of Chuck Siska for the initial version of the ICSP library. The authors would like to acknowledge support from National Science Foundation award number NSF CCR-9806898, from DARPA/ITO DABT63-98-C-004.

References

[1] A. Chowdhary, S. Kale, P. K. Saripella, N. K. Sehgal, and R. K. Gupta. Extraction of functional regularity in datapath crcuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits ans Systems*, September 1999.

[2] D. D. Gajski, R. Domer, and J. Zhu. Ip-centri methodology and design with the specc language. In *System-Level Synthesis*. Kluwer Academic Publishers, 1999.

[3] R. K. Gupta. Timing-driven system design. In *CS Workshop on VLSI*, April 1999.

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

[5] T. Kam, S. Rawat, D. Kirkpatrick, R. Roy, G. S. Spirakis, N. Sherwani, and C. Peterson. Eda challenges facing future microprocessor design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, December 2000.

[6] D. Sylvester and K. Keutzer. Getting to the bottom of deep submicron. In *International Conference on Computer-Aided Design*, 1998.

[7] S. Tarafdar and M. Lesser. A data-centri approach to high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, November 2000.

[8] K. Wakabayashi and T. Okamoto. C-based soc design flow and eda tools: An asic and system vendor perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, December 2000.

[9] Accelera C/C++ Class Library Standardization Working Group home page: http://www.eda.org/alc-wgp.

[10] IEEE/DATC C++ Modeling Standardization Effort home page: http://www.ics.uci.edu/ rgupta/datc/.

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

[12] The Superlog Language home page: http://www.superlog.org/.