Object-Oriented Techniques in Hardware Modeling using SystemC

A. Tsikhanovich, E.M. Aboulhamid,
DIRO, Université de Montréal
CP6128 Centre-Ville
Montréal, Qc, Canada H3C 3J7
Phone: +1(514)343-6822, FAX: +1(514)343-5834
{aboulham, tsikhana}@iro.umontreal.ca

Guy Bois
DGE, École polytechnique de Montréal
C.P. 6079, Succ. Centre-ville
Montréal (Québec), Canada H3C 3A7
Phone: (514) 340-5944, Fax: (514) 343-4147
guy.bois@polymtl.ca

Abstract

Digital systems complexity and short time-to-market characterize today's digital system design. The object-oriented methodology can be applied in order to cope with these problems by allowing reuse, specialization of existing models to express new functionality or modification in implementation and descriptions at different levels of abstraction. In this paper, we illustrate the benefits of such methodology using SystemC.

1. Introduction

The object-oriented (OO) approach has gained great popularity among the software engineering professionals, and became the paradigm of choice for many software product designers. Object technologies lead to reuse of program components, which results in higher-quality programs and faster software development.

Digital systems complexity and short time-to-market characterize today’s digital system design. Since nowadays hardware design is becoming more and more a programming activity, the object technologies can be considered as a way to cope with these challenges. Hardware designers can increase flexibility and reusability of the modeled systems or their components by using OO techniques. The development time and cost can be significantly reduced and the modeling of more complex systems can become more efficient.

SystemC (SC) is the new de-facto standard in modeling hardware designs, it is an open source environment based on C++. In this article, we will present a SystemC OO methodology and explore the results of its application in hardware modeling. In this analysis we will rely on concepts described in [1].

Section 2 presents SystemC OO methodology in terms of examples showing SC and VHDL expressions of object-oriented design strategies and their application to Register Transfer Level (RTL) hardware descriptions and hardware/software (H/S) modeling. Different implementations of the DLX processor models will be considered in the section 3 using the OO methodology. Section 4 will conclude this work.

2. SystemC object-oriented programming

2.1 Object-oriented design strategies

OO programming presents a number of powerful design strategies based on practical and proven software engineering techniques expressed by means of the corresponding object-oriented software programming language structures. These strategies are presented in Table 2-1. They are fundamental and are encountered in different problem-solving contexts and without doubt in hardware design problem-solving context too. Widely used standard hardware description languages like VHDL or Verilog do not support all of the variations of each of these strategies. Therefore the OO programming is not completely available with standard HDLs. We will compare VHDL and SystemC capabilities to illustrate the relevance of these strategies in Hardware and H/S modeling.

<table>
<thead>
<tr>
<th>Design Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstraction</td>
<td>Simplifying the description of a real-world entity to its essentials</td>
</tr>
<tr>
<td>separation</td>
<td>Treating &quot;what&quot; an entity does and &quot;how&quot; it does it independently of each other</td>
</tr>
<tr>
<td>composition</td>
<td>Building complex &quot;whole&quot; systems by assembling smaller &quot;parts&quot; in one of two basic ways: association or aggregation</td>
</tr>
<tr>
<td>generalization</td>
<td>Identifying common elements among different entities in one of four ways: hierarchy, generality, polymorphism, patterns</td>
</tr>
</tbody>
</table>

2.2 Abstraction and Separation.

“Abstraction: A named, tangible representation of the attributes and behavior relevant to modeling a given entity for some particular purpose.” [1]

According to the purpose of the design, a single entity may have many valid abstractions. The widely used abstraction in hardware design is a hardware component or a “primitive” expressed in the two domains:

- Hardware components (e.g., registers, buses, multiplexers)
- Software components (e.g., instructions, algorithms)
• Structural domain: a component is described in terms of an interconnection of more primitive components.
• Behavioral domain: a component is defining by its input/output response [5].

In VHDL, the primary entity is called a design entity; it corresponds to a module in SC. In VHDL only structural domain abstraction can be expressed. SC provides additional types of abstraction: communication abstractions can be expressed through the channels, and the ordinary C++ class concept can be used to express the behavioral abstractions. These additional abstractions are very important in H/S codesign, allowing the designer to commit to a particular H/S partitioning late in the design process. In the pure hardware modeling these abstractions are also important since they enable transaction based modeling which are very efficient regarding simulation performance. They are also important in inheriting specific behaviors and attributes when designing a library of components. CAD houses were forced to describe these libraries outside of VHDL, due to the lack of behavioral abstractions.

“Separation: the independent specification of an interface and one or more implementations of that interface.”

In VHDL, a design entity consists of two different types of descriptions: the interface description and one or more architectural bodies. In hardware design the interface is presented by the description of entity’s inputs and outputs. The architectural bodies can specify the behavior of the entity or a structural decomposition of the entity using more primitive components. In software engineering, an interface defines an external aspect that the entity using more primitive components. In software design contains a bus (which by definition can allow reading or writing), but the instance that we will use is an AMBA bus. The object will be aware that it is an AMBA bus. There are two forms of composition: association and aggregation. In an aggregation, the whole is visible and therefore accessible. In association the interacting parts may be shared by different compositions, as they are externally visible. The association purpose is to allow objects to be connected to each other or to know about each other. The association method of building systems is known as the plug-and-play technique and is widely used in the validation and evaluation of the created system.

2.3 Composition

“Composition: an organized collection of components interacting to achieve a coherent, common behavior.” [1]

There are two forms of composition: association and aggregation. In an aggregation, the whole is visible and therefore accessible. In association the interacting parts may be shared by different compositions, as they are externally visible. The association purpose is to allow objects to be connected to each other or to know about each other. The association method of building systems is known as the plug-and-play technique and is widely used in the validation and evaluation of the created system.

In hardware design the structural design decomposition is a basic modeling technique and that is why the composition is naturally presented by syntactical HDL structures. In VHDL the composition is presented,
in our opinion, in the aggregation form, the design entity is visible as a whole by the others ones. The association form of the composition, which is missing in VHDL can be very useful in the high-level descriptions of the designed system, in test-bench generations, in design exploration and in application of the incremental refinement methodology to a design. The different associations of objects encapsulating some functionality can be validated or verified to find the optimal solution. For example, the association approach can be used in H/S codesign to validate design partition. The different associations of objects presenting hardware and software parts can be constructed and validated to find an optimal decision.

2.4 Generalization.

"Generalization: the identification, and possible organization, of common properties of abstractions." [1]

The generalization identifies commonalities among a set of entities. The commonality may be established in terms of attributes, behavior, or both. The generalization design strategy is directly connected to reusability. There are four forms of generalization expressed by object-oriented software structures:

- Hierarchy;
- Genericity;
- Polymorphism;
- Patterns;

We skip design patterns form of generalization applied to hardware modeling, since it is well detailed in [7].

2.4.1 Hierarchy

Hierarchy organizes the abstraction commonalities into a tree structure, where the root of any sub-tree contains all the attributes and behavior common to every descendant of that root.

"Hierarchy: an organization of abstractions into a directed graph in which the arcs denote an “is-a” relation between a more generalized abstraction and the one or more specializations derived from it." [1]

In VHDL there are no syntactical structures to express this kind of hierarchy; therefore reuse is limited to the component level, i.e. designers must use the component as is or have to design a new one.

OO and SC allow the next step beyond the structural view. We can use the behavioral hierarchy (inheritance) concepts to generalize a hardware primitive interface (Figure 2.1). This would be similar in representing the relationship in VHDL between architecture and an entity. However we have a more general hierarchy here, subclasses may inherit methods and attributes, have default behaviour that they can be overridden totally or partially. Inheritance provides the potential for building new solutions to problems by adding incremental capability to existing problem solutions through sub-classing. This concept is one of the most fundamental OO features to enable reuse.

2.4.2 Genericity

Genericity expresses the abstractions commonalities with using parameters. The different specializations can be constructed by parameter variations.

"Genericity: a named generalization expressed in terms of other unspecified abstractions that are denoted by parameters." [1]

In VHDL genericity is presented by means of generics that pass the instance-specific information other than port connections to an entity. Generics are used to parameterize an entity, making it general-purpose.

VHDL's generics are a particular case of SC genericity mechanisms. We can pass them as a parameter to the class constructor or we can use the notion of templates. This notion is very powerful and has been used extensively in the OO programming. For example, let us consider the following SC declarations:

```sc
sc_in<bool> X;  (a)
sc_uint<32> tdata;  (b)
sc_out<sc_uint<32> > data; (c)
```

(a) X is of class `sc_in` (input port) with the `bool` as a template. The class `sc_in` is a templated class, and objects of that class can be instantiated later as Boolean, Integers or any other class of SC. (b) Declares a variable which is an unsigned integer of 32 bits. In this case, templates allow integers or fixed point numbers of any precision, which is vital in DSP applications. (c) Illustrates the power of templates: data belongs to the class of output port with the specialization to the class of 32 bit unsigned integers. Most of these features are absent in VHDL.

2.4.3 Polymorphism

"Polymorphism: the ability to manipulate objects of distinct classes using only knowledge of their common properties without regard for their exact class."

The polymorphism enables new classes possessing required properties to be handled by a polymorphic algorithm. In software engineering, as a rule, an interface defines the object behavior. In digital design a hardware component behavior can be seen as a collection of processes that implement concurrent operation of digital systems. The polymorphism expresses the ability of an operation to take on different forms when applied to different objects. The polymorphism applied to the HDL processes gives an enormous potential of reuse enabling
the incremental behavior refinement of particular hardware modules.

In OO systems the polymorphism is implemented by a mechanism called dynamic binding or late binding. In VHDL the mechanism of late binding is used by configurations that allow design descriptions to be managed during simulation. SC offers similar an other capabilities by the mechanism of process polymorphism. It is possible to override the particular processes and in that way create the specialized components fully exploiting the existing code. For example, this will allow us to describe a generic ALU and then derive from more specialized ALUs for floating point manipulation. The higher level models will be aware that they are using ALUs in general and each particular instance will interpret a generic action in its own way.

C++ offers two types of polymorphism; one implemented by virtual functions, named execution polymorphism, and the second, implemented by C++ templates, named compilation polymorphism or parameterized polymorphism [6]. These two approaches are very close to each other because they allow expressing an algorithm once and applying it to different types. In our context, we have considered the “process polymorphism”, the ability of the simulation kernel to connect the appropriate implementation of behavior to a derived class. This mechanism may allow us the reuse of a test-bench, and apply it to different target objects (or implementations).

3. A case study: Pipelined DLX processor

Initially we have implemented the DLX with a five stage integer pipeline; each stage was described by an SC module (Figure 3.1). We then extended the basic pipeline model of DLX, by adding the implementation of the floating-point operations. Since DLX_CPU is a structural module, we have modified its structure to incorporate instantiations of the modules, implementing the floating-point operations. We specialized ID, EX and MEM stage classes to capture modifications concerning the addition of new operations. It was implemented using the “process polymorphism” concept (Figure 3.2).

Using the association form of generalization to model the pipeline, we can simulate two DLX objects, one of which containing refined components, and validate the behavior of designing system. There is also the possibility to simulate and compare the different implementations of designed system using the same test-bench.

Figure 3.1 Structural model of DLX processor

Figure 3.2 DLX CPU extension implementing floating-point operations using Tomasulo algorithm

4. Conclusion

Current design methodologies are not sufficient to cope with new requirements such as design reuse and time to market; OO methodologies are a strong candidate to resolve these problems. This paper objective was to familiarize the designers with these new paradigms in the hardware domain, and to show their direct application in modeling reuse, design space exploration, verification reuse, and simulation performance. In our opinion, these concepts have to be understood to overcome a new productivity gap.

Synthesis constitutes another challenge; we should define a global methodology for modeling and synthesis in order to avoid the pitfalls of VHDL where the semantics of a model differ from simulation to synthesis and worse from one level of synthesis to the other. This remains an open research direction in OO methodologies.

References