Application of Software design patterns to DSP library design

Pontus Åström
Dept. of Electroscience
Lund University
P.O Box 118
SE-221 00 Lund, Sweden
Pontus.Astrom@es.lth.se

Stefan Johansson
Turin Networks
Petaluma, CA, USA
SJohansson@turinnetworks.com

Peter Nilsson
Dept. of Electroscience
Lund University
P.O Box 118
SE-221 00 Lund, Sweden
Peter.Nilsson@es.lth.se

ABSTRACT

The design of a hardware data path library is one of the harder problems in design for reuse. Thanks to the appearance of hardware modeling libraries based on C++, it is possible to apply advanced software techniques to design such a library. This paper shows how software design patterns can be applied to hardware design. Design patterns yield a twofold advantage: a faster design process, and a library that is more extensible and modular than an equivalent HDL counterpart. From a VHDL–C++ design comparison we have found that those factors might result in a reduction of the code size by a factor of two.

1. INTRODUCTION

The software community has been the source for many of the larger productivity leaps in hardware design for the last two decades. Two of the major transfers have been procedural programming and object-oriented design (OOD). Procedural programming appeared with the advent of the HDLs Verilog and VHDL, both derived from the procedural software languages C respectively ADA. Several proposals have been given for how to add support for the OOD methodology to hardware design [1]. Another recent idea to benefit from the strengths of the object-oriented paradigm is to move the hardware design to a suitable software language like C++ [2, 3].

Hardware design in C++ is promising as it simplifies the application of good software design techniques to hardware designs. Design by reuse of *design patterns* is one such technique. It will be shown here that software design patterns can be applied to hardware designs.

A design pattern can be thought of as the skeleton of a good design that can be reused in new designs in different fields and over time. In hardware design typical examples of design patterns are finite state machines (FSM) or communication protocols. With the availability of a public pattern catalog with good and well proven patterns, design by reuse of design patterns will boost the productivity in several areas. It will result in faster and less error prone design processes. It will give designs that are more modular and extensible. Finally, it results in a higher abstraction level as the design can be explained and understood in terms of the applied patterns.

Currently there exist very few public design patterns aimed specifically for hardware design. However, in the software domain and particularly in the OOD area, design patterns have had a big impact over the last few years. Several public software pattern catalogue exist [4, 5, 6] and a large productivity boost has been observed from pattern usage [7].

It will be shown that several design patterns from the software pattern catalog [4] easily can be applied to the design of a hardware data path library in C++, designed with aid of the library Ocapi [3]. In total four patterns have been applied. The patterns are: Composite, Object Adaptor, Abstract Factory and Decorator.

The usability of those patterns is demonstrated by the design of a synchronizer for a mobile radio system. The design is compared to an equivalent VHDL design. The results show that the C++ design has half the code size compared to the VHDL design.

2. MOTIVATION FOR USING DESIGN PATTERNS

Every experienced designer/programmer reuses structures and designs that have worked well in the past more or less without thinking. Previously, the reused patterns were seldom put in print, the patterns therefore were spread very slowly or not at all. Learning patterns was thus a question of getting experience, to recognize the patterns that repeat over time. Unfortunately, gathering experience takes long time.

By writing down good design patterns in a structured way that clearly states the intention, usage and implementation, much is won in terms of reuse, learning and abstraction. Three advantages can be seen:

- Novice designers get a much steeper learning curve. They can quickly acquire the experience gathered by expert programmers over many years to write robust, well partitioned and error free code.
- Design discussions can be performed at a much higher level of abstraction. Instead of discussing how indi-

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

vidual classes should be arranged, the discussion can revolve around whether a particular pattern is a good way to structure a given problem.

- Better documentation. It can be written in terms of patterns to give the reader a direct understanding of how the code is organized.

A study of applying patterns in the software domain have verified the above points [7]. We can not see any fundamental differences between the software and hardware design processes, that would invalidate the results for hardware designs. However, the advantages will not show up until there is a public design catalog and an educated community that can communicate in terms of patterns.

3. HARDWARE DESIGN IN C++

Hardware design has essentially become a software activity. For many projects the hardware part is minor in terms of code size and work force. As there are benefits in a uniform development environment for hardware-software co-design, there is a motivation to move hardware design to a software language like C++. However, the move can be motivated for hardware-only designs were the corresponding simulation model is written in a software language. A few of the advantages are:

- Promotes design by stepwise refinement. Small hardware units can fast and easily be verified in the simulation program by simply being exchanged with the corresponding software units. This speeds up the development process as the dedicated test benches need to test fewer cases.
- Higher abstraction level due to the object oriented features of C++.
- Access to several useful software libraries including the standard C and C++ libraries.
- Simplified usage of several well proven design paradigms including the mentioned object oriented paradigm with use of design patterns.

There are disadvantages in designing with C++ too. The larger part of the hardware design community is still used to traditional HDL such as VHDL and Verilog. We believe however that moving to C++ offers a set of design techniques that matches well up against the learning curve of moving to another language.

4. THE TOOL OCAPI

Ocapi [3, 8] is a C++ tool aimed for design and simulation of hardware according to the data flow paradigm. Ocapi provides an object-oriented design library for data path designs and takes full advantage of the language C++.

With Ocapi it is feasible to encapsulate a data path component with a finite state machine (FSM) completely in a C++ class. Ocapi defines a bus object `busTp` (Tp for type) for transport of fixed point data between two data path objects. In hardware the object busTp translates into plain wiring.

It is easy to raise the abstraction level by defining new bus types. Bus types for complex numbers and vectors are used by the data path library. With the container classes of the standard template library (STL) [9] it is easy to define the desired buses.

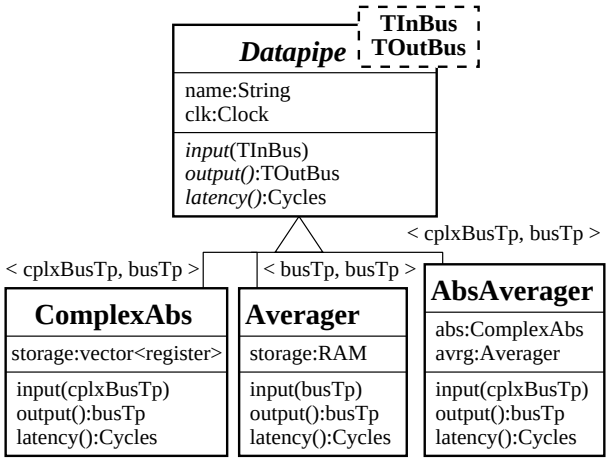

Figure 1: The data path storage composition.

```
typedef std::pair<busTp, busTp > cplxBusTp;
typedef std::vector<busTp >       vecBusTp;
```

5. THE DATA PATH LIBRARY

The library is intended for data path design and matured in parallel with a custom DSP implementation. Currently the library contains about 20 data path objects ranging from simple FIFOs to a complete channel synchronizer[10]. All data path objects are currently in the hardware mapped algorithm form. This kind of data paths are fast, have a high throughput and require little control. The drawback is that the data path objects might have low utilization and thus the silicon usage might be excessive. In section 5.2 it is shown how the data path library can be extended to better adapt to different requirements.

5.1 The repository

The composite pattern is chosen as base to arrange the data path objects in a tree-like, hierarchical structure. This arrangement has two benefits. First, it allows the data path objects to be treated uniformly with no difference between leaf objects designed from scratch and compositions of objects. Second, the user can easily extend the library by defining new composite objects from the existing leafs and composites.

Figure 1 shows a small example of the composite pattern in UML (Unified Modeling Language) notation[11]. The root class is named *Datapipe*, it defines the interface for the data path objects. Three concrete data path objects are inherited form the root class.

The Datapipe class is an abstract base class that defines the type of the data path objects, i.e. the names of the method calls, the input and return types of those calls and so on. Three abstract methods are defined. `latency():Cycles` returns the delay in clock cycles from input to output, `input(TInBus)` and `output():TOutBus` connects the data path object to an input and an output bus. `TInBus` and `TOutBus` are parameters, or templates in C++ notation,

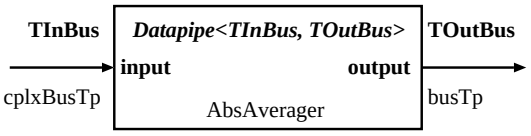

Figure 2: A structural view of a data path component.

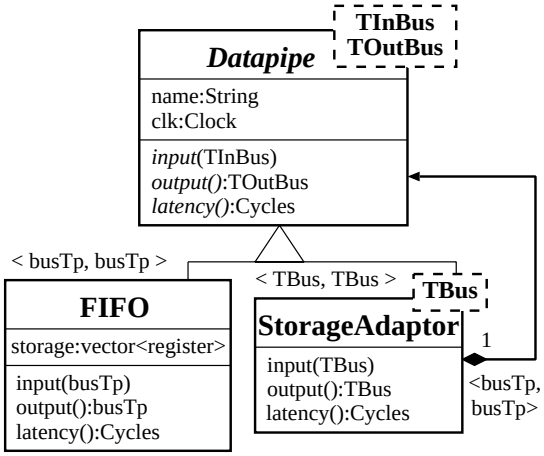

Figure 3: Application of the adaptor to the composite pattern.

whose exact types is specified in each subclass. For example, the `ComplexAbs` object specifies the type of `TInBus` to `cplxBusTp` and the type of `TOutBus` to `busTp`.

The concrete data path objects derived from the class Datapipe are of two kinds, leafs and composites. Leafs are designed from scratch while composites are compositions of leafs and other composites. In figure 1 `ComplexAbs` and `Averager` are leafs and the `AbsAverager` object is a composite. `AbsAverager` is composed of the data path objects `ComplexAbs` and `Averager`.

The interface defined by the Datapipe class maps to a component with a structural view as shown in Figure 2 with one input and one output. Text in boldface describes the component as defined by the Datapipe class. Each concrete class that inherits from the Datapipe class maps the parameterized types `TInBus` and `TOutBus` to real bus types. `AbsAverager` is used as an example. It maps the input and the output to the bus types `cplxBusTp` respectively `busTp`.

5.2 Interface adaptation

The library defines several data path objects related to storage which all have the input and output type `busTp`. Thus, storage objects only store signed or unsigned integers. There are reasons to be able to store data of other types as well, for instance complex and vector data of the types `cplxBusTp` and `vecBusTp`. The *object adaptor* pattern can be used to change the interfaces of an existing object without touching it. In Figure 3 the adaptor StorageAdaptor that adapts a given type to `busTp` is shown. Within the adaptor a mapping to `busTp` is defined for each other bus type. As an example, the mapping from `cplxBusTp` is shown in

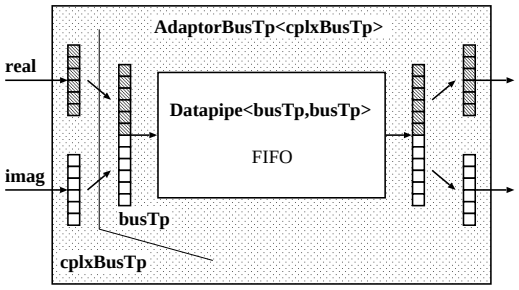

Figure 4: A structural view of the adaptation of `cplxBusTp` to `busTp`.

Figure 4. In this particular example the adaptee is a FIFO. The corresponding code to instantiate a FIFO for complex numbers and connect it to two buses is shown below.

```
1  cplxBusTp inBus, outBus;
2  Adaptor<cplxBusTp> cplxFIFO
3    ( new FIFO( FIFOsize ));
4  cplxFIFO.input( inBus );
5  outBus = cplxFIFO.output();
```

The adaptor is instantiated on lines 2 and 3. The constructor takes a pointer to the adaptee as an argument, which here is a newly created instance of FIFO. At the instantiation the whole data path required for the adaptation is created. The data path is connected to external buses on lines 4 and 5.

The advantage with this solution is that only one adaptor object has to be implemented. If it is desirable to utilize the storage objects for new bus types the only code that has to be written is a new bus type map within StorageAdaptor.

5.3 Dynamic data path creation

The current implementation of the library is targeted to one particular DSP implementation with a specific need of high throughput. To be useful for a broad range of DSP implementation it has to support other optimization constraints as well. One extension has been designed to allow the library to better adapt to varying requirements of different designs. The extension utilizes the pattern *Abstract factory*. It is used to implement a similar structure to the entity–architecture–composition structure in VHDL.

Consider the design of an IP data path block utilizing a module that can be designed according to different tradeoffs. Which version to choose depends partly on end user requirements that not are known at the design time of the IP block. What is needed is a mechanism to let the user decide the implementation of some modules within the IP block.

A pattern that matches the specification is the Abstract Factory pattern. A possible implementation with two concrete factories for low power and small area respectively, is shown in Figure 5. The abstract data path object `ComplexAbs` has one implementation targeted for each concrete factory. In the figure the client is the IP-block. The user of the IP-block gives a concrete factory as argument at the instantiation of the IP-block/client. In this case it is either a High-Throughput or a LowPower factory. Depending on which, either the Cordic or the LockUp module gets instantiated in the IP-block.

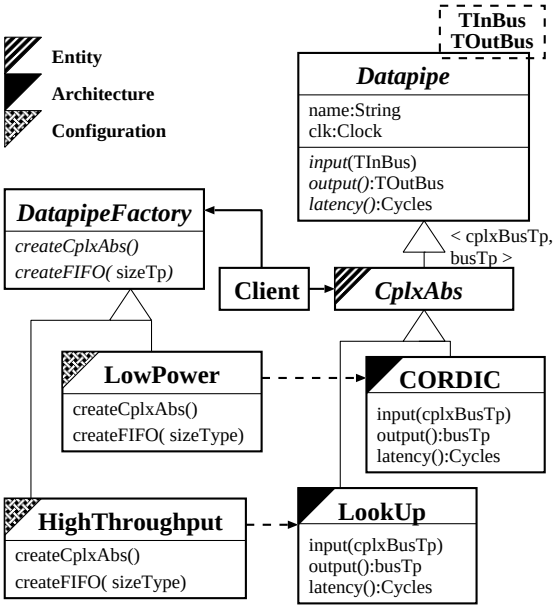

Figure 5: The abstract factory pattern. Triangular shaded areas indicate the corresponding structure in VHDL.

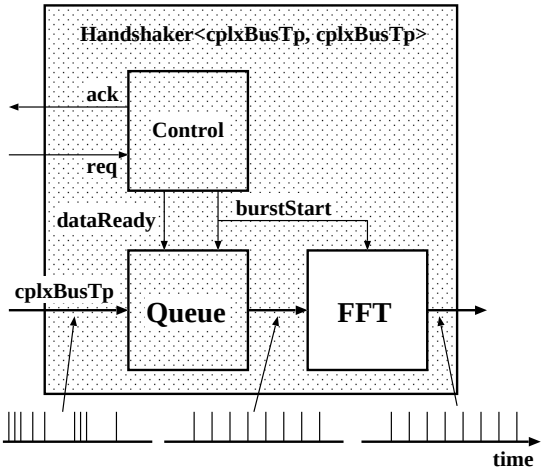

Figure 6: The decorator pattern that implements the handshaking function. The time axis below gives an idea of the appearance of data in the different buses.

Note the close ties to the VHDL composition in an entity, an architecture and a configuration. The entity corresponds to the `ComplexAbs` object, the architectures are the concrete classes inheriting from `ComplexAbs` and finally the configurations corresponds to the concrete DatapathFactory objects which connects an interface to an implementation. The design shows that particular design features of the current HDLs not have to be lost by a move to an object-oriented design environment like C++.

5.4 Handshaking

Consider the problem of connecting a data path object, processing data in blocks, like a FFT, to a source that cannot guarantee a continuous data stream, like a multitasking processor. This communication channel requires a handshaking protocol and a data queue to buffer incoming data as shown in Figure 6.

The *Decorator* pattern provides a object composition for this problem. The idea of the pattern is to decorate the data path object with the extra functionality, in his case the handshaking protocol. In Figure 7 the object structure for the handshaking decorator is shown. The buses ack and req are used by the handshaking protocol. The Queue class buffers incoming data and forwards it to the decorated data path object when a whole block of data has been received.

The advantage of this pattern is that only one implementation is needed to add handshaking to all different block processing data path objects.

6. EVALUATION OF LIBRARY

The library has been evaluated with a frame synchronization algorithm for an Orthogonal frequency division multiplexing (OFDM) mobile radio system [10]. In Figure 8 the data flow of the algorithm is shown. As the algorithm is hardware mapped this is also the the structure of the hardware implementation.

Table 1: Code sizes for VHDL respectively C++ implementation of the synchronizer.

Synchronizer	# lines	# tokens
C++	936	3607
VHDL	2100	8200

The design was chosen for two reasons. First it is an algorithm that is well suited for implementation with the library because it is data path centric. Second, and most important, an existing VHDL implementation of the algorithm can be used to compare the performances of the C++ and VHDL based designs respectively. The VHDL version has been implemented in hardware. The C++ version has been verified at VHDL level. The same algorithms were used for all sub blocks. For the Rect2Pol and the Rotate blocks, the CORDIC algorithm was chosen.

It is important to stress here that the comparison only compares hard facts like line and token count. Soft facts like reusability, ease of design and robustness cannot be measured objectively by design. However, those are of out most importance for the success of a design paradigm. The relative merits of the soft facts must therefore to the larger part be judged from the arguments given. Only to a minor extent do they appear as measurable hard facts in a design of this comparatively small size.

In table 1 the total number of lines and the token count is given for both implementations. It can be seen that the C++ code is roughly half of the size. It is true for the token count as well which is a more accurate estimate that the line count as it does not depend on coding style to the same extent. We believe that the results in the table are fair for this kind of design.

The smaller code size in the C++ design is mainly due to two factors. First, the number of reusable components was

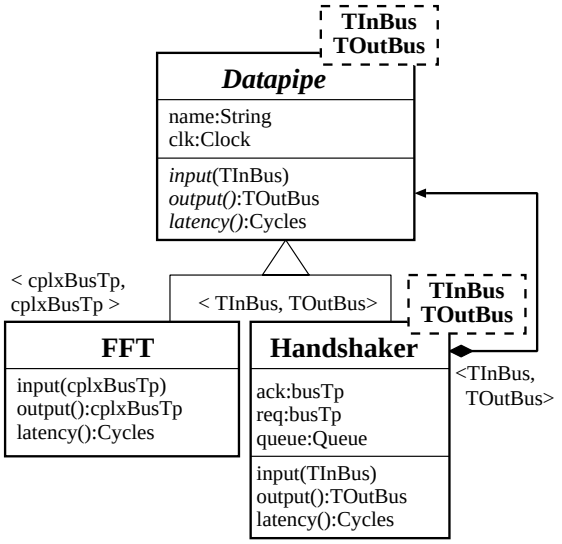

Figure 7: The handshaking decorator.

higher due to the well structured object-oriented library. Second, C++ as language allows for a more compact description compared to VHDL.

7. CONCLUSIONS

It has been shown that several software design patterns with minor modifications, can be applied to the design of a hardware data path library. The usage of design patterns is beneficial as the result is structures that are modular, reusable, robust and require little overhead. As shown only one implementation is required for each of StorageAdaptor and Handshaker to add the desired functionality.

All software patterns cannot be used in hardware design, however they can provide insight into how good structures are designed, and thus, act as inspiration in the design of dedicated hardware patterns.

We have shown by an example applying the described patterns that a C++ design is roughly half in code size compared to a VHDL design. We do not however claim that code size is an adequate measure of the merits of the presented library. Several important properties of a library like usability, reusability, ease of use and structure can not be objectively measured by a rough figure like code size. However, we believe it gives a hint about the performance that can be expected.

8. ACKNOWLEDGMENTS

This work is founded by the Integrated Electronic Systems Program (INTELECT) of the Swedish Foundation for Strategic Research (SSF).

9. REFERENCES

[1] S. Liao, S. Tjiang, and R. Gupta, "An efficient implementation of reactivity for modeling hardware in the scenic design environment," in *Proc. of the 34th Design Automation Conference*, June 1997, pp. 70–75.

[2] SystemC, http://www.systemc.org.

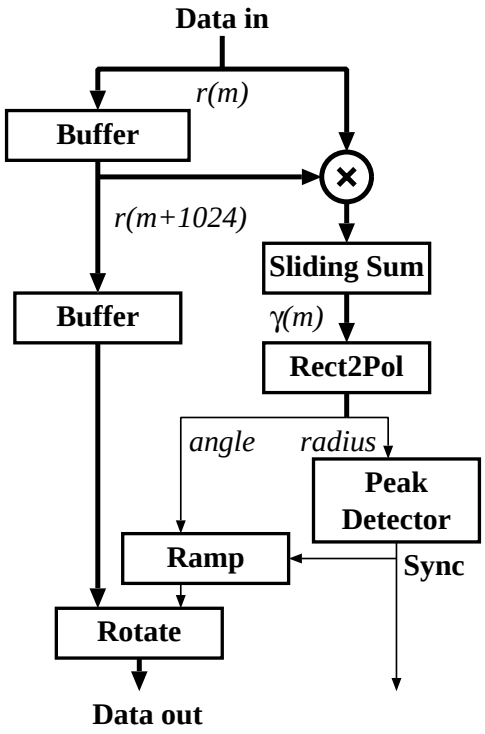

Figure 8: The data flow view of the synchronizer. Bold wires are complex numbers. Thin wires are scalars except form the sync wire which is boolean.

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

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[5] J. Coplien and D. Schmidt, *Pattern Languages of Program Design*, Addison-Wesley, 1995.

[6] J. Vlissides, J. Coplien, and N. Kerth, *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.

[7] K. Beck, J. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulish, and J. Vlissides, "Industrial experience with design patterns," in *Proc. of 18th International Conference on Software Engineering*, March 1996, pp. 103 –114.

[8] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, and I. Bolsens, "Hardware reuse at the behavioral level," in *Proceedings of the Design Automation Conference*, 1999, p. 784.

[9] M. Austern, *Generic Programming and the STL*, Addison-Wesley, 1999.

[10] S. Johansson and P. Nilsson, "Hardware implementation of an ofdm syncronization algorithm," in *Proc. of Midwest symposium on Circuits and Systems*, Las Cruces, New Mexico, August 1999.

[11] M. Fowler, *UML Distilled*, Addison-Wesley, 1997.