Manuscript Number:

Title: Aspect-oriented RTL HW design using SystemC

Article Type: Research paper

Keywords: aspect-oriented programming; digital hardware design; hardware description languages; SystemC

Corresponding Author: Mr. Tiago Rogério Mück, M.Sc.

Corresponding Author's Institution: Federal University of Santa Catarina

First Author: Tiago Rogério Mück, M.Sc.

Order of Authors: Tiago Rogério Mück, M.Sc.; Antônio A Fröhlich, PhD

Abstract: With the increasing complexity of digital hardware designs, hardware description languages are being pushed to higher levels of abstraction, thus allowing for the use of design artifacts which were previously exclusive to the software domain. In this paper we aim to contribute to this scenario by proposing artifacts and guidelines for digital hardware design using object-oriented and aspect-oriented programming concepts. Our methodology is based on features provided by SystemC, a C++-based hardware description language, and leverages on its synthesizable subset in order to produce designs suitable for circuit synthesis. Our experimental results show that our design artifacts provide an increased level of flexibility and reusability while increasing the final circuit size by only 2%.

Tiago Rogério Mück and
Prof. Dr. Antônio Augusto Fröhlich
Federal University of Santa Catarina
Software/Hardware Integration Laboratory
PO Box 476, 88040-900 FFlorianópolis, SC, Brazil
Phone: +55 48 3721-9516
E-Mail: guto@lisha.ufsc.br
WWW: http://lisha.ufsc.br/∼guto

March 21, 2013

Microprocessors and Microsystems
Editor-in-Chief
Prof. Dr. Lech Jóźwiak

Dear Prof. Jóźwiak,

We would like to submit the attached manuscript, "Aspect-oriented RTL HW design using SystemC", for consideration for possible publication in the *Microprocessors and Microsystems: Embedded Hardware Design.*

In the manuscript, we describe a SystemC-based aspect-oriented approach for creating more reusable RTL hardware designs. In summary, its main contributions are:

- It provides a comprehensive analysis of related work in the area of AOP applied to hardware design;
- It describes an AOP-based method for designing synthesizable hardware components; the proposed mechanisms are based only on standard C++ features available on the SystemC synthesizable subset;
- It presents the design and implementation of real world case studies, which allowed us to elucidate the tradeoffs of AOP applied to hardware; In our cases, we have increased the code reuse by about 33% with a circuit area overhead of only 2%.

The manuscript has not been published or accepted for publication and it is not under consideration at another journal.

Sincerely,

Tiago Rogério Mück and Antônio Augusto Fröhlich

Highlights

- We describe an AOP-based method for designing synthesizable hardware components using SystemC;
- Our mechanisms are based only on standard C++ features available on the SystemC synthesizable subset;
- No external aspect weaving tool is required;
- In a PABX case study, we have increased the code reuse by about 33% with a circuit area overhead of only 2%.

Aspect-oriented RTL HW design using SystemC

T. R. Mück[1,*], A. A. Fröhlich[1]

Federal University of Santa Catarina, Florianópolis, Brazil

Abstract

With the increasing complexity of digital hardware designs, hardware description languages are being pushed to higher levels of abstraction, thus allowing for the use of design artifacts which were previously exclusive to the software domain. In this paper we aim to contribute to this scenario by proposing artifacts and guidelines for digital hardware design using object-oriented and aspect-oriented programming concepts. Our methodology is based on features provided by SystemC, a C++-based hardware description language, and leverages on its synthesizable subset in order to produce designs suitable for circuit synthesis. Our experimental results show that our design artifacts provide an increased level of flexibility and reusability while increasing the final circuit size by only 2%

Keywords: aspect-oriented programming, digital hardware design, hardware description languages, SystemC

1. Introduction

The complexity of digital hardware design is increasing as the advances of the semiconductor industry allow for the use of sophisticated computational resources in a wider range of applications. This new deployment scenario is leading to a growing interest in high-level methodologies for digital hardware design. Solutions and methodologies that have been successfully deployed in the scope of large-scale software systems, such as *object-oriented programming* (OOP), are being introduced to hardware design as well. An example of a *hardware description language* (HDL) which supports OOP is SystemC, a C++-based HDL [1].

[*]Corresponding author

Email addresses: `tiago@lisha.ufsc.br` (T. R. Mück), `guto@lisha.ufsc.br` (A. A. Fröhlich)

[1]Authors full address:
UFSC/CTC/LISHA
PO Box 476
88040-900 Florianópolis - SC - Brazil
Phone/Fax: +55 48 3721-9516

HDLs are used to create descriptions of electronic circuits which can be used either for design verification or for hardware synthesis. Differently from most software programming languages, HDLs are intrinsically parallel and provide explicit means for describing timing. VHDL [2] and Verilog [3] are the most widely used HDLs for *register transfer level* (RTL) design. In RTL, circuits are described in terms of the operations between storage elements which are synchronized using clock signals. In this level, a hardware design may be composed by several *modules*, whose instances are interconnected to build the system. However, in contrast to software programming languages, module communication occurs through timed protocols and signals defined by the module input/output interface instead of function calls.

The use of OOP-capable HDLs (e.g. SystemC) enables an increased level of flexibility and reusability in hardware design. Object-oriented features, such as inheritance and polymorphism, allow the stepwise refinement and generalization of hardware IPs in a way it is not possible with standard HDLs. However, analogous to software, in hardware some system-wide cross-cutting concerns still cannot be elegantly encapsulated. For example, in complex circuits, interconnection of several entities is realized by introducing buses. A bus physically interacts with other components (e.g. CPU, DMA ...), but it is difficult to use a module or an OOP class to encapsulate the bus because its interface and arbitration method have to be implemented in every attached component [4]. Other examples of cross-cutting concerns in hardware design can be found in parts of a system related to its overall functionality or the implementation of non-functional properties (e.g. fault-tolerance and low-power mechanisms, hardware debugging through scan chains, clock handling, ...) [5]. Even by using OOP in hardware, this scattered code is hard to maintain and bugs may be easily introduced. This motivates the introduction of *aspect-oriented programming* (AOP) [6] techniques for hardware design.

AOP is an elaboration over OOP to deal with crosscutting concerns. AOP proposes the encapsulation of these concerns in special classes called *aspects*. An aspect weaving semantic is then defined to describe when and how aspects are applied to components. The aspect weaving is defined in terms of *pointcuts* and *join points*. The latter defines which points of a program can be affected by an aspect, while the former defines when and how aspects are inserted at the join points. Some extensions to OOP languages have been proposed to support these concepts. For example, AspectJ [7] and AspectC++ [8] extend Java and C++ with full support for AOP features. They provide both new language constructs and an aspect weaving tool that applies aspects to the base code before it is processed by the traditional compiling chain.

As can be seen in the next section, several works have already shown that the introduction of the aforementioned concepts to hardware design is expected to provide means for the encapsulation of cross-cutting concerns and an increase in the overall design quality. However, previous works in this topic have focused on high-level specification and AOP features are used mostly for code instrumentation and verification [9, 10, 11]. Research targeting the use of AOP for the actual design of hardware functionalities have proposed language constructs and features which are not supported by hardware synthesis tools [5, 12, 13, 14], thus lacking a more comprehensive discussion about the physical overhead related to the use of AOP in hardware.

In order to contribute to this scenario, in this paper we describe a digital hardware design method which leverages on SystemC features in order to enable the implementation of hardware components using OOP and AOP concepts. We propose the use of a design strategy that yields components in which its dependencies from the execution scenario are encapsulated as *aspects* and *configurable features*. The proposed aspect weaving semantics are implemented using only standard C++, hence eliminating the need of extra tools and compilers. Additionally, such features are within the SystemC synthesizable subset [15], thus yielding *synthesizable components*. Our method is finally illustrated by the design and implementation of case studies extracted from an industrial *Private Automatic Branch Exchange* (PABX) application.

In summary, this work has the following contributions:

- It provides a comprehensive analysis of related work in the area of AOP applied to hardware design.
- It describes an AOP-based method for designing synthesizable hardware components.
- It presents the design and implementation of real world case studies, which allowed us to elucidate the tradeoffs of AOP applied to hardware.

The remaining of this paper is organized as follows: section 2 presents a discussion about related work; sections 3 and 4 presents our design methodology and its evaluation through case studies; and section 5 closes the paper with our conclusions.

2. Related work

Several works have already addressed the use of AOP concepts for hardware design. *Engel and Spinczyk* [4] discussed the nature of crosscutting concerns in VHDL-based hardware designs and proposed a hypothetical AOP extension for VHDL. In *Bainbridge-Smith and Park* [16] the authors discussed how the separation of concerns may relate to different levels of algorithmic abstraction. They have mentioned the development of ADH, a new HDL based on AOP, but further details about ADH are not mentioned. *Burapathana et al.* [17] proposed the use of AOP concepts to sequential logic design. Nevertheless, they focused on very simple and low level examples like flip-flops and logic gates.

There are also several works which proposed the use of AOP concepts mostly for hardware verification. *Kallel et al.* [9] proposed the use of SystemC and AspectC++ to implement assertion checkers. The authors focused on the verification of *transaction-level models* (TLM) [18] in which transaction state updates are used as pointcuts. They provide a framework in which the user's verification classes extend base aspect classes that implement the pointcuts and the verification primitives. In *Vachharajani et al.* [11] the authors developed the *Liberty Structural Specification Language* (LSS). In LSS each module can declare instances which emit certain events at runtime. These events behave like pointcuts of AOP. Each time a certain state is reached or a value is computed, the instance will emit the corresponding event and user-defined aspects will perform statistics calculation and reporting. *Liu et al.* [10] also proposed AOP-based instrumentation, but focusing on high-level power estimation. They have developed a methodology based on SystemC in which AspectC++ is used to define special power-aware aspects. These aspects are used as configuration files to link power-aware libraries with SystemC models.

Other works provide AOP features not only for verification, but also for the actual design of hardware. *Déharbe and Medeiros* [14] present and assess possible applications of AOP in the context of integrated system design by using SystemC with AspectC++. Differently from the works discussed previously, they showed how AOP can be used to encapsulate some functional characteristics of hardware components. They modeled as aspects the replacement policy of a cache, the data type of an FFT, and the communication protocol between modules. However, only simulation results are shown and they do not compare the implementation of aspect-based components against components with all the functionalities hand-coded. In a similar work, *Liu et al.* [12] implemented a SystemC model for a 128-bit floating-point adder and described the implementation of the same model using AOP techniques. But, synthesis results are not provided and the two models are compared only in terms of functionality to show that the AOP design works like the original SystemC-only design. ASystemC [5] also extends SystemC in a similar fashion, but, instead of using AspectC++, authors developed their own aspect weaver. The new aspect language was introduced through different case studies involving high-level estimation of circuit size, feature-configurable products, and assertion-based verification. However, the evaluation of ASystemC also does not compare hardware generated using traditional methods with their proposal. AspectVHDL [19], on the other hand, defines an aspect weaving approach that can yield synthesizable code. However, using VHDL as the base language limits the potential for reusability since OOP features are not supported.

Other works in this area follow different approaches. The *e* programming language [20] was designed for modeling and verification of electronic systems and some of its mechanisms can be used to support AOP features. Apart from its OOP features, *e* has some constructs to define the execution order of overloaded methods in inherited classes, which can be used to define pointcuts and implement aspects. Indeed, this can be used to implement the behavior of hardware components, but *e* is more focused in high-level specification and there is not any tool support for synthesis. *Jun et al.* [13] analyzed the application of *Aspectual Feature Module* (AFM) [21] to HDLs. They have implemented a RISC processor using SystemC and FeatureC++, and showed how AFM enables the incremental development of hardware through the modularization of code fragments for the implementation of a function. However, AOP is used only for encapsulation of verification code and the authors do not provide synthesis results of the resulting code.

3. Designing hardware components using scenario adapters and configurable features

Previous works that explored the use of AOP in SystemC have relied on AspectC++. We have also based our approach on methodologies which have been used in the software domain. The *Application-driven Embedded System Design* (ADESD) [22] methodology elaborates on commonality and variability analysis—the well-known domain decomposition strategy behind OOP—to add the concept of aspect identification and separation at early stages of design. Throughout ADESD's domain engineering process, the properties that transcend the scope of single abstractions are captured as *aspects*. Such aspects include mostly dependencies from the execution scenario and non-functional properties. This enables components to be reused on different scenarios with the application of proper *aspects*. This aspect weaving is performed by constructs called *Scenario adapters* [23].

Whether such guidelines can also be defined for designing hardware has not yet been investigated, but nonetheless, SystemC enables the introduction of convenient C++ constructs to increase the quality of hardware designs. This will be demonstrated in the next sections.

3.1. Scenario adapters

Scenario adapters were developed around the idea of components getting *in* and *out* of an execution scenario, allowing actions to be executed at these points, therefore, a scenario must define at least two different operations: *enter* and *leave*. These actions must take place respectively before and after each of the component's operation in order to setup the conditions required by the scenario. For example, in a compressed scenario, *enter* would be responsible to decompress the component's input data, while *leave* would compress its outputs. Figure 1 shows the general structure of a scenario adapter. The *Scenario* class represents the execution scenario and incorporates, via aggregation, all of the aspects which define its characteristics. Then, the adaptation of the component to the scenario is performed by the *Scenario Adapter* class using inheritance. Using classical AOP terms, there is a *join point* before and after each method of a component's public interface, while the *pointcuts* are defined by the scenario adapter.

In order to bring these ideas to hardware, the difference between hardware and software must be considered. In the software domain, components are objects which communicate using method invocation (considering an OOP-based approach), so the scenario adapters were originally developed to provide means to efficiently wrap the method calls to an object. However, in a RTL design, components have input and output signals instead of a method or function interface. Notwithstanding, SystemC provides features to separate the implementation of the communication interface from the behavior, thus allowing us to use scenario adapters in way very similar to the one used in software.

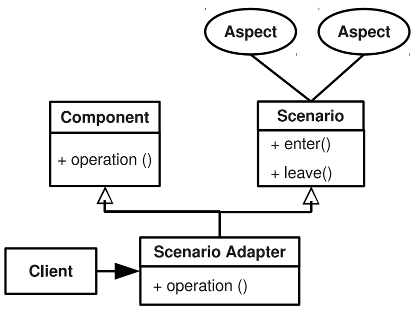


Figure 1: UML-based class diagram showing the general structure and behavior of a scenario adapter.

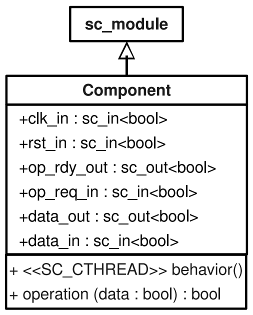
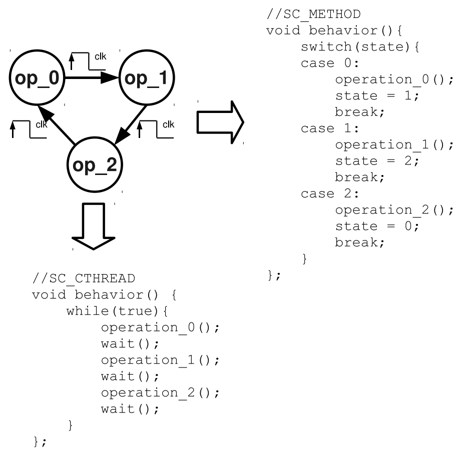


Figure 2: SystemC description of a RTL component with a simple handshaking interface.

Figure 2 shows a SystemC implementation of a component with the interface separated from the behavior. SystemC defines hardware components by the specialization of the *sc_ module* class. Components communicate using special objects called *channels*. SystemC channels can be used to encapsulate complex communication protocols at register transfer or higher levels of abstraction. However, these complex channels lie outside the SystemC synthesizable subset, so we use only use *sc_ in* and *sc_ out* channels, which define simple RTL input and output ports for components. The entry point for the component execution must be a method defined as SystemC processes (*Component::behavior*). The example below shows how this method can be used for implementing the handshaking protocol (using channels *op_ reg_ in* and *op_ req_ out*) and calling the method that implements the operation (*Component::operation*):

```
...
If (op_req_in) {
    op_rdy_out = 0;
    data_out.write(operation(data_in.read()));
    op_rdy_out = 1;
}
...
```

Apart from the signal-oriented interface, the scheduling of operations among clock cycles is another important characteristic that differentiates a RTL SystemC implementation of a component from its analogous software implementation in C++. SystemC allows for different styles to define timing. In our components we use SystemC's clocked threads (*SC_ CTHREAD*), in which all operations are synchronized to a clock signal using *wait()* statements. All operations defined between two *wait()* statements occur in the same clock cycle. By using *wait()* statements one can describe the synchronization more clearly without the need of implementing explicit state machines. Figure 3 shows this difference. It compares the synchronization of three operations that must be executed in a loop in different clock cycles. The rightmost implementation uses a *SC_ METHOD* process, which provides an implementation style very similar to VHDL/Verilog, while the leftmost shows the implementation using *SC_ CTHREAD*.



```
//SC_METHOD
void behavior(){
    switch(state){
    case 0:
        operation_0();
        state = 1;
        break;
    case 1:
        operation_1();
        state = 2;
        break;
    case 2:
        operation_2();
        state = 0;
        break;
    }
};
```

```
//SC_CTHREAD
void behavior() {
    while(true){
        operation_0();
        wait();
        operation_1();
        wait();
        operation_2();
        wait();
    }
};
```

Figure 3: A state machine implemented using *SC_ CTHREAD* and *SC_ METHOD*.

By using these constructs, one can describe components susceptible for adaptation using scenario adapters. However, other differences between software and RTL hardware design must be considered. In software, the execution model is naturally sequential. A software OOP model may contain several independent classes and still have a single and sequential execution flow, unless the designer explicitly models parallelism. On the other side, in RTL hardware all modules runs in parallel, and the operations inside the modules are also executed in parallel unless the designer explicitly schedules them in different clock cycles. Taking this into account, we propose to define each aspect as a single and independent hardware component. Figure 4 shows this definition.

The aspects can use the same handshaking mechanisms described previously in order to make it easier to synchronize their execution with the rest of the design. The class *Aspect_ Common* encapsulates this protocol as shown below:

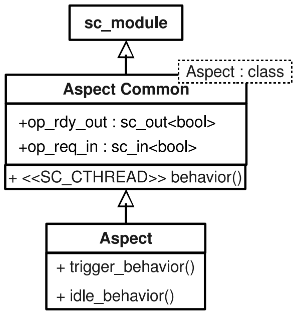


Figure 4: Definition of aspects as a SystemC module.

```
If (op_req_in) {
    op_rdy_out = 0;
    static_cast<Aspect*>(this)->trigger_behavior();
    op_rdy_out = 1;
}
else {
    op_rdy_out = 1;
    static_cast<Aspect*>(this)->idle_behavior();
}
```

Aspect_ Common defines basic input/outputs signals and a SystemC process on which the aspect behavior is going to execute. Each class defining an aspect inherits from *Aspect_ Common* and must implement at least two methods: *Aspect::trigger_ behavior* defines the behavior executed when an operation is triggered; and *Aspect::idle_ behavior* defines the behavior executed when the aspect is in an idle state. Notice that the aspect classes derive from template instantiations of *Aspect_ Common* using themselves as template parameters. This is known as *Curiously Recurring Template Pattern* (CRTP) [24], and we use it to avoid the use of dynamic polymorphism, which is not supported for hardware synthesis. The final form of the scenario adapter in hardware can be seen in Figure 5 (for simplicity, some details already shown in Figures 2, 3, and 4 are omitted).

The *Scenario* class represents the execution scenario and incorporates, via aggregation, all of the aspects which define its characteristics. It defines *enter* and *leave* methods to encapsulate the implementation of the handshaking protocol which trigger the aspects. The piece of code below shows how the scenario's *enter* operation could be implemented:

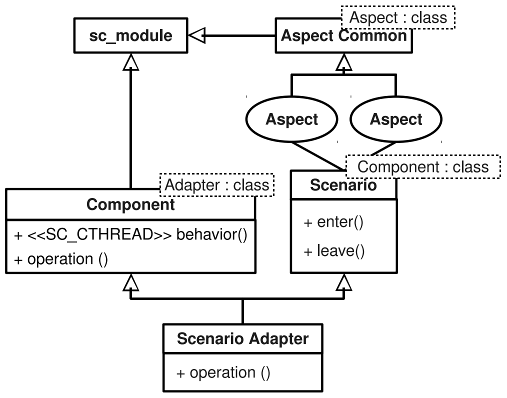


Figure 5: UML-based diagram of the final scenario adapter. Input/output signals are omitted for simplicity.

```
while(op_rdy_0 &  ... & op_rdy_n)
    wait();

for all aspects {
    ...
    //set aspects inputs/outputs
    ...
    op_req_0 = 1;
    ...
    op_req_n = 1;
}
wait();
for all aspects {
    op_req_0 = 0;
    ...
    op_req_n = 0;
}
```

All aspects are triggered at the same time and executes in parallel, however, if required by a component, each aspect can be executed sequentially in a specific order. This modification can be performed by specializing the scenario for specific components. The piece of code below illustrates this specialization. The first class definition is a default implementation of *Scenario*, while the remaining ones define specific implementations for different components.

```
template<class T> Scenario {...};

template<> Scenario<Component_0> {...};
...
template<> Scenario<Component_n> {...};
```

The adaptation of the component to the scenario is performed by the *Scenario_ Adapter* class via inheritance. The methods that implement the operations of the component are overridden in the *Scenario_ Adapter* class, which wraps the original methods with calls to the scenario's *enter* and *leave*:

```
Scenario::enter();
Component::operation();
Scenario::leave();
```

Similarly to the implementation of the aspects, CRTP is used to avoid dynamic polymorphism. Calls to operations inside *Component::behavior* are performed using the same approach described for the *Aspect_ Common* class:

```
static_cast<Adapter*>(this)->operation();
```

3.2. Configurable features

Additionally to the aspect separation, several characteristics can be identified by the designer as configurable features of the components. Such characteristics represent fine variations within a component, which can be set in order to change slightly its behavior or structure.

Special template classes called *Traits* are used to define which characteristics of each component is activated. The code sample below shows the implementation of a trait class and how it is referred inside an operation:

```
template <> struct Traits<Component>
{
    static const bool feature_1   = true;
    static const bool feature_2   = false;
    ...
    static const bool feature_n   = true;
};

...

void Component::operation(){
    ...
    if (Traits<Component>::feature_1) {
        ...
    }
    ...
}
```

Additional behavior is executed inside *Component::operation* if the feature *feature_ 1* is enabled. Since the condition in the *if statement* can be statically evaluated, the additional code is completely optimized away by the synthesis tool when the condition is false.

Metaprogramming techniques [25] can be associated with the trait concept to also modify the structure through inheritance. The example below shows how one can define the base class of *Component* as a configurable feature:

```
public Component :
    public sc_module,
    public
    IF<Traits<Component>::feature_n,
        Base_Class_1,
        Base_Class_2>::Result
{
   ...
};
```

The *IF* metaprogram uses partial template specialization to return one of two specified types based on a boolean value. Its implementation is shown below:

```
template<bool condition, typename Then, typename Else>
struct IF
{ typedef Then Result; };

template<typename Then, typename Else>
struct IF<false, Then, Else>
{ typedef Else Result; };
```

Figure 6 shows how this concept can be represented in a UML diagram. The component definition indicates its features. In the case of conditional inheritance, both possibilities are represented with a special notation at the relationship definition.

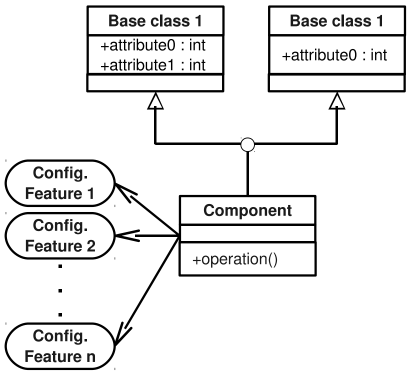


Figure 6: UML representation for configurable features

3.3. ADESD and classic AOP

Several previous works have already discussed aspect-oriented hardware design using SystemC and proposed solutions based on classic AOP concepts using AspectC++. Apparently, AspectC++ provides more powerful mechanisms for aspect implementation then ADESD's scenario adapters, especially when it comes to the definition of pointcuts. Scenario adapters were not designed to add behavior in specific points inside an operation. However, it is possible to circumvent this limitation using other standard OOP and the proposed configurable features. For example, in *Déharbe and Medeiros* [14] the replacement policy of a memory cache and the data type of an FFT are encapsulated as aspects, while a straightforward alternative would be to use inheritance and templates parameters.

It is also worth mentioning that the implementation of ADESD's mechanisms can be realized using only standard C++/SystemC features. No extensions to the language or an aspect weaving tool are required. Previous works focused on tools and mechanisms that were deployed originally for software development (e.g. AspectC++), therefore limiting its use for the generation of synthesizable hardware. For example, AspectC++ may introduces dynamic pointers and objects that are outside the SystemC synthesizable subset[15], thus allowing the development of simulation-only models. Although simple aspects in AspectC++ lead to simple code transformation, previous works showed no evidence that AspectC++ woven code can be synthesized. This limits the evaluation of the physical overheads (e.g. silicon area and performance) associated to AOP.

As can be seen in the following sections, the use of OOP, scenario adapters, and configurable features yield synthesizable code, thus enabling the use of ADESD's mechanisms in all levels of the design process.

4. Evaluation

In this section we describe the experimental results of the evaluation of our approach. First, we describe the components which are part of our case studies along with the scenario aspects that we have identified. Then, we provide an evaluation of the designs, considering reusability and hardware synthesis results for both area and delay.

4.1. Case studies

We have analyzed a PABX application from one of our industry partners and designed some of its basic building blocks using the proposed design artifacts. A PABX system is basically a commutation matrix that switches connections amongst different input/output data channels. These channels are connected to phone lines (through an AD/DA converter), tone generators, and tone detectors. The system also provides the transmission of voice data through an Ethernet network to support *voice over IP* (VoIP) lines. The basic components defined as case studies are described in more details below. All of them were designed according to the guidelines described in section 3.

Resource scheduler: the PABX system is implemented over the *Embedded Parallel Operating System*(EPOS) [26]. This case is a hardware implementation of the EPOS's thread scheduler [27]. A scheduler may perform operations both synchronously (upon request by another component) or asynchronously (by preempting the execution of another component). This complexity makes it a good case study. Furthermore, a hardware-implemented scheduler reduces jitter and improves the support for real-time applications. Such characteristics can provide a major impact in the PABX system, since each channel in the PABX central is handled in software by a different thread.

FIR filter: *finite impulse response* (FIR) filters are some of the most frequently used and well-known blocks in digital signal processing , thus offering a high reuse potential in a wide range of application domains. This motivated a careful and flexible design. The main feature of our filter is the support to both complex and real arithmetic by using configurable feature, as shown in Figure 7. The interface and MAC core are implemented separately and inherited according to the feature set.

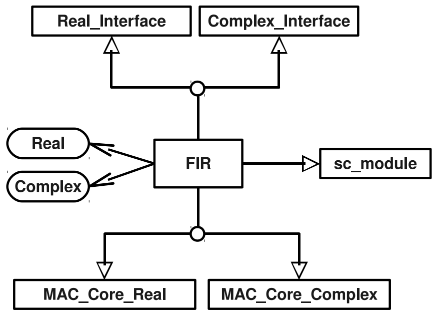


Figure 7: FIR filter structure

DTMF detector: a *Dual-Tone Multi-Frequency* (DTMF) detector [28] is responsible for detecting DTMF tones in the phone lines. It is a critical component in any PABX system, thus was selected as a case study.

4.2. Scenario aspects

We have identified two different execution scenarios in our application domain: a *debugged* scenario and a *compressed* scenario in which on-chip debugging and compression of sampled data are required, respectively. Figure 8 shows the aspects implemented for each scenario and their interface. The aspects inherit the basic handshaking protocol from *Aspect_ Common* (section 3.1) and implement only their specific interface and operations. More details about the scenarios and its aspects are given below.

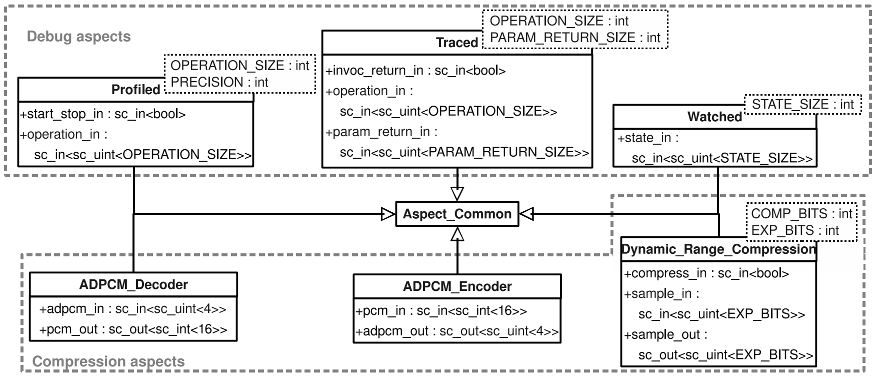

Debug aspects
Profiled
OPERATION_SIZE : int
PRECISION : int
+start_stop_in : sc_in<bool>
+operation_in :
sc_in<sc_uint<OPERATION_SIZE>>
Traced
OPERATION_SIZE : int
PARAM_RETURN_SIZE : int
+invoc_return_in : sc_in<bool>
+operation_in :
sc_in<sc_uint<OPERATION_SIZE>>
+param_return_in :
sc_in<sc_uint<PARAM_RETURN_SIZE>>
Watched
STATE_SIZE : int
+state_in :
sc_in<sc_uint<STATE_SIZE>>
Aspect_Common
Dynamic_Range_Compression
COMP_BITS : int
EXP_BITS : int
+compress_in : sc_in<bool>
+sample_in :
sc_in<sc_uint<EXP_BITS>>
+sample_out :
sc_out<sc_uint<EXP_BITS>>
ADPCM_Decoder
+adpcm_in : sc_in<sc_uint<4>>
+pcm_out : sc_out<sc_int<16>>
ADPCM_Encoder
+pcm_in : sc_in<sc_int<16>>
+adpcm_out : sc_out<sc_uint<4>>
Compression aspects

Figure 8: The debug and compression scenarios implemented

Debugged scenario: the upper part of Figure 8 shows the aspects that can be part of a debugged scenario. Unlike previous works, which focused mostly on simulation-time tracing and logging, we have focused on *design for testability* [29] and implemented aspects for on-chip debugging using a JTAG scan chain. The implemented aspects define the following debugging functionalities: *Watched* causes the state of a component to be dumped every time it is modified; *Traced* causes every operation execution to be signalized; and *Profiled* counts the number of clock cycles used by the component for each operation. The width of debugging signals within each aspect is defined as template parameter in order to match the requirements of different components.

Compressed scenario: the lower part of Figure 8 shows the aspects that can be part of a compressed scenario. These aspects provide means to compress digital signals, thus reducing the number of bits required to either store or transmit them. The *Dynamic_ Range_ Compression* aspect provides operations to compress or expand the dynamic range of a signal (i.e. the largest and smallest possible values of a signal) using a linear transformation. The *ADPCM_ Encoder/Decoder* aspects implement the same operations using an *adaptive differential pulse-code modulation* (ADPCM) algorithm [30] to convert 16-bit samples to 4-bit samples.

4.3. Scenario adapters implementation

Figure 9 shows how we have applied the aspects using scenario adapters (for simplicity, details such as methods, ports, and some hierarchies are omitted). The highlighted components are the scenario adapters, which applies the scenario aspects to the components, yielding the following new cases:

Debugged scheduler: the *Scheduler_ Adapter* class implements the scenario adapter for our scheduler. It inherits from the *Scheduler* and *Debugged_ Scenario* classes, adding support for on-chip debugging. The *Debugged_ Scenario* class incorporates the aspects *Profiled*, *Traced*, and *Watched*.

Debugged FIR filter with dynamic range compression: the *FIR_ Adapter* class adapts *FIR* for both *Debugged_ Scenario* and *Compressed_ Scenario*. The *Compressed_ Scenario* class incorporates the aspects *Dynamic_ Range_ Compression*, *ADPCM_ Encoder*, and *ADPCM_ Decoder*. This scenario exemplifies a situation when scenario specialization is required, since both ADPCM and dynamic range compression may not be used by the same component. In this case, the specialization of *Compressed_ Scenario* for *FIR* incorporates only dynamic range compression. The final implementation compresses the samples before the filter, and expands them afterwards, thus trading-off precision for resource consumption.

Debugged DTMF detector with ADPCM: the *DTMF_ Detector_ Adapter* follows the same approach of *FIR_ Adapter*. However, in this case study, the specialization of *Compressed_ Scenario* incorporates only the ADPCM aspects.

4.4. Experimental results

We have evaluated the overhead introduced by our artifacts in our case studies by comparing the aspect-oriented implementation described previously with an implementation in which the aspect's behavior is *hand-coded* in the

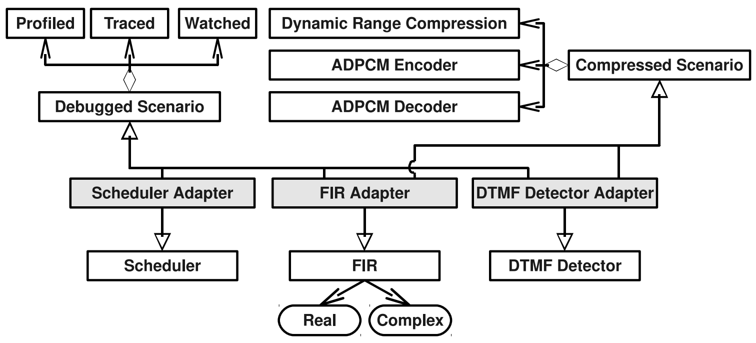


Figure 9: Overview of the aspect-oriented design. The highlighted components are the scenario adapters implemented as case studies.

core components. To evaluate the efficiency, we have synthesized the designs to a physical circuit in a *field-programmable gate array* (FPGA). To evaluate the reusability we have analyzed the number of lines of code in each case study.

4.4.1. Reusability

Table 1 compares the number of lines of code of the scenario adapters with the hand-coded versions of all case studies. Line *Scenarios-adapted* shows the number of lines for each design artifact in the scenario-adapted design. Line *Hand-coded* shows the number of lines when our approach is not used and the aspects are hand-coded in the core components. The results show that the number of lines written is actually smaller in the handed-coded designs. However, this only highlights the initial development overhead of providing a modular implementation of aspects and scenarios. In the scenario-adapted implementations, about 33% of the written code is reused in at least two of our three case studies; while the hand-coded implementation shares no common code. The code from the *debugged scenario* and the related aspects is reused across all case studies, while the code from the *compressed scenario* is partially reused since different specializations of this scenario are used in two cases.

Table 1: Number of lines of code of all cases

		Lines of code
Scenario-adapted	Base	976
	Aspects	317
	Scenarios	210
	Adapters	155
	Total	1573
Hand-coded		1332

4.4.2. Synthesis results

We have synthesized our design to physical circuits targeting an FPGA and analyzed their performance and size (area). Figure 10 shows the synthesis flow. The SystemC designs are first converted to VHDL descriptions using Celoxica's Agility 1.3. Then, Xilinx ISE 13.1 is used for both logic synthesis and place-and-route (the process of fitting a circuit for a specific FPGA device). As our target device we have chosen a Xilinx Virtex6 XC6VLX240T FPGA.

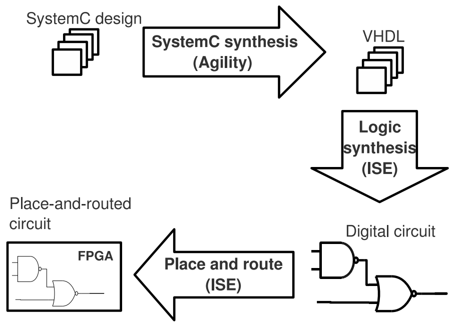


Figure 10: Design synthesis flow targeting FPGAs

Table 2 shows the number of *slices* used (a configurable logic element in Xilinx's FPGA) and the *longest path delay* (LPD) of the three case studies. The LPD represents the performance of the circuit, while the number of slices is used to evaluate the area. The results show that the use of scenario adapters yields a very low overhead in terms of both resource consumption and performance. The average area of the scenario-adapted components is about 2% higher than the hand-coded components. This small overhead comes basically from the additional signal and registers required by the handshaking protocol that is used to trigger the aspects, which is not required when everything is coded within a single SystemC module. The average difference in performance is also small (0.78%). Curiously, in the first case study, the scenario-adapted design has a smaller LPD. This may be the result of some optimization algorithm applied in the place-and-route backend.

Table 2: FPGA synthesis results. Cases 1-3 refer to the scheduler, FIR, and DTMF detector, respectively.

Case study	Scenario-adapted		Hand-coded		Difference	
	Slices	LPD(ns)	Slices	LPD(ns)	Slices	LPD
Case 1	651	10.30	645	10.42	0.93%	-1.15%
Case 2	133	07.93	130	07.70	2.30%	2.98%
Case 3	139	11.46	135	11.40	2.96%	0.52%

5. Conclusion

This paper has introduced an AOP-based method for designing hardware components using SystemC. The components dependencies from different specific execution scenarios were successfully encapsulated and further applied to the core components through the use of scenario adapters and configurable features. By using the proposed design strategy, on-chip debugging features were implemented as separated aspects and fully reused throughout the case studies, thus showing the ease-of-use and versatility of scenario-adapters to handle homogeneous crosscutting concerns.

The adaptation of components that require the addition of different behaviors in the same execution scenario was also demonstrated. We have used partial template specialization to create a compressed scenario that incorporates either dynamic range compression or ADPCM coding depending on the component it is applied to. Although a full reusability could not be achieved in this case, the scenario specialization mechanism provides a straightforward and clear way to encapsulate heterogeneous crosscutting concerns.

In addition, we have also focused in the design of synthesizable hardware components, rather than verification and simulation-only models. The experimental results showed that our design artifacts can not only increase reusability but also be efficiently synthesized to physical hardware. The scenario adapter structure and handshaking mechanisms for scenario activation increased the circuit size by only 2%, while introducing a negligible overhead in the circuit performance.

References

[1] P. R. Panda, SystemC: a modeling platform supporting multiple design abstractions, in: Proceedings of the 14th international symposium on Systems synthesis, ISSS '01, ACM, New York, NY, USA, 2001, pp. 75–80.

[2] IEEE, Std 1076-2000: IEEE Standard VHDL Language Reference Manual (2000).

[3] IEEE, Std 1364-2001: IEEE Standard Verilog Hardware Description Language (2001).

[4] M. Engel, O. Spinczyk, Aspects in hardware: what do they look like?, in: Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software, ACP4IS '08, ACM, New York, NY, USA, 2008, pp. 5:1–5:6.

[5] Y. Endoh, ASystemC: an AOP extension for hardware description language, in: Proceedings of the tenth international conference on Aspect-oriented software development companion, AOSD '11, ACM, New York, NY, USA, 2011, pp. 19–28.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: Proceedings of the European Conference on Object-oriented Programming'97, Vol. 1241 of Lecture Notes in Computer Science, Springer, Jyväskylä, Finland, 1997, pp. 220–242.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An Overview of AspectJ, in: Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01, Springer-Verlag, London, UK, UK, 2001, pp. 327–353.

[8] O. Spinczyk, A. Gal, W. Schröder-Preikschat, AspectC++: an aspect-oriented extension to the C++ programming language, in: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications, CRPIT '02, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2002, pp. 53–60.

[9] M. Kallel, Y. Lahbib, R. Tourki, A. Baganne, Verification of systemc transaction level models using an aspect-oriented and generic approach, in: Design and Technology of Integrated Systems in Nanoscale Era (DTIS), 2010 5th International Conference on, 2010, pp. 1 –6.

[10] F. Liu, Q. Tan, X. Song, N. Abbasi, AOP-based high-level power estimation in SystemC, in: Proceedings of the 20th symposium on Great lakes symposium on VLSI, GLSVLSI '10, ACM, New York, NY, USA, 2010, pp. 353–356.

[11] M. Vachharajani, N. Vachharajani, D. I. August, The liberty structural specification language: a high-level modeling language for component reuse, in: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, PLDI '04, ACM, New York, NY, USA, 2004, pp. 195–206.

[12] F. Liu, O. A. Mohamed, X. Song, Q. Tan, A case study on system-level modeling by aspect-oriented programming, in: Proceedings of the 2009 10th International Symposium on Quality of Electronic Design, IEEE Computer Society, Washington, DC, USA, 2009, pp. 345–349.

[13] Y. Jun, L. Tun, T. Qingping, The application of Aspectual Feature Module in the development and verification of SystemC models, in: Specification Design Languages, 2009. FDL 2009. Forum on, 2009, pp. 1 –6.

[14] D. Déharbe, S. Medeiros, Aspect-oriented design in systemC: implementation and applications, in: Proceedings of the 19th annual symposium on Integrated circuits and systems design, SBCCI '06, ACM, New York, NY, USA, 2006, pp. 119–124.

[15] OSCI, SystemC Synthesizable Subset Draft 1.3 (2010).
URL `http://www.systemc.org/`

[16] A. Bainbridge-Smith, S.-H. Park, ADH: an aspect described hardware programming language, in: Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on, 2005, pp. 283 – 284.

[17] P. Burapathana, P. Pitsatorn, B. Sowanwanichkul, An Applying Aspect-Oriented Concept to Sequential Logic Design, in: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02, ITCC '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 819–820.

[18] L. Cai, D. Gajski, Transaction level modeling: an overview, in: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '03, ACM, New York, NY, USA, 2003, pp. 19–24.

[19] M. Meier, S. Hanenberg, O. Spinczyk, Aspectvhdl stage 1: the prototype of an aspect-oriented hardware description language, in: Proceedings of the 2012 workshop on Modularity in Systems Software, MISS '12, ACM, New York, NY, USA, 2012, pp. 3–8. doi:10.1145/2162024.2162028.
URL `http://doi.acm.org/10.1145/2162024.2162028`

[20] M. Vax, Conservative aspect-orientated programming with the e language, in: Proceedings of the 6th international conference on Aspect-oriented software development, AOSD '07, ACM, New York, NY, USA, 2007, pp. 149–160.

[21] S. Apel, T. Leich, G. Saake, Aspectual Feature Modules, IEEE Trans. Softw. Eng. 34 (2008) 162–180.

[22] A. A. Fröhlich, Application-Oriented Operating Systems, no. 17 in GMD Research Series, GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.

[23] A. A. Fröhlich, W. Schröder-Preikschat, Scenario Adapters: Efficiently Adapting Components, in: Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, USA, 2000.

[24] J. O. Coplien, Curiously recurring template patterns, C++ Rep. 7 (1995) 24–27.

[25] K. Czarnecki, U. W. Eisenecker, Generative programming: methods, tools, and applications, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[26] The EPOS Project, Embedded Parallel Operating System (2012).
URL `http://epos.lisha.ufsc.br/`

[27] H. Marcondes, R. Cancian, M. Stemmer, A. A. Fröhlich, On the Design of Flexible Real-Time Schedulers for Embedded Systems, in: Proceedings

of the 2009 International Conference on Computational Science and Engineering - Volume 02, CSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 382–387.

[28] Z. Xinyi, The FPGA Implementation of Modified Goertzel Algorithm for DTMF Signal Detection, in: Proc. of the 2010 International Conference on Electrical and Control Engineering, Wuhan, China, 2010, pp. 4811–4815.

[29] T. Williams, K. Parker, Design for testability–A survey, Proceedings of the IEEE 71 (1) (1983) 98–112.

[30] ITU-T, ITU-T Recommendation G.726: 40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM) (1990).
URL `http://www.itu.int`

Tiago Rogério Mück received his M.Sc. in Computer Science from the Federal University of Santa Catarina, Brazil in 2013. Since 2010, he has been a researcher at the Laboratory for Software and Hardware Integration at the same university. His current research interests include computer architecture and embedded systems.

Antonio Augusto Fröhlich received his Ph.D. in Computer Science from the Technical University of Berlin in 2001. He has been a professor in the Computer Science Department, Federal University of Santa Catarina, Brazil since 1995 and head of the Laboratory for Software and Hardware Integration since 2001. His current research interests include embedded systems and operating systems.

Click here to download high resolution image

Click here to download high resolution image