Unified Design of Hardware and Software Components

Journal:	*Transactions on Computers*
Manuscript ID:	Draft
Manuscript Type:	Regular
Keywords:	C.3.d Real-time and embedded systems < C.3 Special-Purpose and Application-Based Systems < C Computer Systems Organization, D.2.3 Coding Tools and Techniques < D.2 Software Engineering < D Software/Software Engineering, B.0 General < B Hardware

Unified Design of Hardware and Software Components

Tiago Rogério Mück and Antônio Augusto Fröhlich

Abstract—The increasing complexity of current embedded systems is pushing their design to higher levels of abstraction, leading to a convergence between hardware and software design methodologies. In this paper we aim at narrowing the gap between hardware and software design by introducing a strategy that handles both domains in a unified fashion. We leverage on *aspect-oriented programming* and *object-oriented programming* techniques in order to provide *unified* C++ descriptions of embedded system components. Such unified descriptions can be obtained through a careful design process focused on isolating aspects that are specific of hardware and software scenarios. Aspects that differ significantly in each domain, such as resource allocation and communication interface, were isolated in *aspect programs* that are applied to the unified descriptions before they are compiled to software binaries or synthesized to dedicated hardware using *high-level synthesis* tools. Our results show that our strategy leads to reusable and flexible components at the cost of an acceptable overhead when compared to software-only C/C++ and hardware-only C++ implementations.

Index Terms—System-level design, HW/SW co-design, High-level synthesis, Aspect-oriented system design.

1 Introduction

Embedded systems are becoming increasingly complex as the advances of the semiconductor industry enable the use of sophisticated computational resources in a wider range of applications. Depending on the application's requirements (e.g. performance, energy consumption, cost, etc.), the development of such systems may encompass an integrated hardware and software design that can be realized by several different architectures, ranging from simple 8-bit microcontrollers to complex *multiprocessor system-on-chips* (MPSoCs).

In order to deal with this complexity, embedded system designs are being pushed to the *system-level*. In this scenario, a convergence between hardware and software design methodologies is desirable, since a unified modeling approach would enable one to take decisions about hardware/software partitioning later in the design process, maybe even automatically. In the last few years, advances in *electronic design automation* (EDA) tools are allowing hardware synthesis from high-level, software-like descriptions. This process is known as *high-level synthesis* (HLS) and allows designers to describe hardware components using languages like C++, and higher-level techniques, such as *Object-Oriented Programming* (OOP). The focus of these tools [1], [2], [3], [4], [5], however, is hardware synthesis, and they do not provide a clear design methodology for developing components which could be implemented as both hardware and software.

Aiming to narrow this gap and to move towards a real unified design approach, in this paper we describe some design guidelines and mechanisms which support the implementation of both hardware and software components from a single C++ description. Our guidelines are built upon the *Application-driven Embedded System Design* (ADESD) methodology [6]. ADESD leverages on OOP and *aspect-oriented programming* (AOP) concepts, defining a domain engineering strategy which allows us to clearly separate the core behavior and the structure of a component from aspects that must be handled differently whether a component is implemented as hardware or software. Such specific characteristics are modeled in special constructs called *aspects* and are applied to the unified descriptions of components only after the hardware/software partitioning is defined, yielding the final implementation in the target domain (hardware or software). In order to generate descriptions that can be efficiently synthesized by HLS tools or compiled to a software binary, the implementation of both the components and the mechanisms which adapt them make extensive use of C++ *generative programming* [7] techniques such as *static metaprogramming*. To evaluate the efficiency of our approach, we present the implementation of a *Private Automatic Branch Exchange* (PABX) SoC of which some components were implemented using our unified design strategy.

It is important to recall that it is not a goal of this work to discuss the quality of hardware implementation generated using HLS tools, neither the intrinsic differences between software and hardware that cannot yet be fully handled by such tools. We take in consideration the fact that these tools impose limitations to source descriptions (such as dynamic allocation and binding). As it will be demonstrated in the following sections, limitations of this kind can be circumvented by our unified approach.

The remaining of this paper is organized as follows: section 2 presents a discussion about works related to the integration of the hardware and software design flows;

• *T. R. Mück and A. A. Fröhlich are with the Software/Hardware Integration Lab , Federal University of Santa Catarina, Florianópolis, Brazil. E-mail: {tiago,guto}@lisha.ufsc.br*

section 3 presents an overview of previous contributions upon which this work is built; in section 4 we discuss the characteristics that are part of software and hardware scenarios and present our approach for their proper separation and implementation; section 5 describes our case studies and shows our experimental results; and section 6 closes the paper with our conclusions.

2 Related work

Several design methodologies and tools were proposed in order to provide more tightly coupled hardware and software design flows. Most of these methodologies are based on the concept of building a system by assembling pre-validated components. The *Ptolemy extension as a Codesign Environment* (PeaCE) [8], an extension of the Ptolemy project [9], is a synthesis framework for real-time multimedia applications. PeaCE takes models composed by synchronous data-flow graphs and extended *finite state machines* (FSMs) and either generates C code or maps the models to pre-existing IP cores and processors. ROSES [10] is a design flow which automatically generates hardware, software, and interfaces from an architectural model of the system and a library of pre-validated components. The *Metropolis* [11] and its successor *Metro-II* [12] follow the *Platform-based design* (PDB) [13] methodology. They propose the use of a metamodel with formal semantics to capture designs. An environment to support simulation, formal analysis, and synthesis is also provided. Despite providing integration frameworks for the entire design flow, these methodologies do not define clear guidelines to design new reusable components. Also, mapping the application model to pre-existing components limits hardware/software partitioning.

In order to overcome these limitations, one must focus on closing the *design gap* of software and hardware components. State-of-the-art EDA tools (e.g. Calypto's CatapultC [1], Synopsys' Synphony [2], Cadence's C-to-Silicon [3], Forte's Cynthesizer [4], Xilinx's AutoESL [5]) support hardware synthesis from high-level C++/C-based constructs. Indeed, several works have already demonstrated the applicability of HLS for implementing hardware components such as signal processing filters, cryptographic cores, and other computationally intensive components [14]. Our aims are different however. We want to describe components in a high level of abstraction, but those should be implementable in hardware as well as in software, and with performance close to software-only and hardware-only implementations.

On this track, the OSSS+R methodology [15] raises the level of *register transfer level* (RTL) SystemC by adding new language constructs to support synthesizable polymorphism and high-level communication. However, hardware/software partitioning must still be done in the beginning of the design process [16], and the inclusion of non-standard language constructs reduces the compatibility of the design with available compilers and hardware synthesis tools. The *Saturn* [17] design flow also contributes in this scenario, but follows a different approach. It aims to close the gap between UML-based modeling and the execution of the models for their verification. The authors have elaborated over *SysML*, an extension of UML for system-level design [18], and developed a tool which generates C++ for software and RTL SystemC for hardware. Although using a single language, software and hardware are still modeled separately. Additionally, it is not clear whether their tool generates code only for the interface and integration of components, or the behavior is also inferred from the SysML models.

SystemCoDesigner [19] is a tool which integrates a HLS flow with design space exploration. The design entry of SystemCoDesigner is an actor-based data flow model implemented using SystemMoC [20]. After design space exploration, actors of this model can be converted to synthesizable SystemC or to C++ for software compilation. However, as the authors themselves claim, SystemCoDesigner targets mostly data-flow-based applications, and they do not provide directions towards a more general deployment. The System-on-chip environment (SCE) [21] takes SpecC models as input and provides a refinement-based tool flow. Guided by the designer, the SCE automatically generates a set of *Transaction-level models* (TLM) [22] that are further refined to pin- and cycle-accurate system implementation.

Other works focus mainly on the interface between software and hardware. The *HThread* project [23], the ReconOS [24] and the BORPH operating system [25] provide a unified interface for both domains. In these works a task performed in hardware is seen with the same semantics as a software thread, and a system call interface is provided to hardware components. However, despite providing this unified interface, they do not cover the gap that still exists between the way the hardware and software threads themselves are implemented. The work of *Rincón et al* [26] is based on concepts from distributed object platforms such as CORBA and Java RMI and provides a tool which generates the communication glue necessary so that hardware and software components can communicate seamlessly.

3 Application-driven Embedded System Design

The ADESD [6] 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. It defines a domain engineering strategy focused on the production of families of scenario-independent components. Scenario dependencies observed during domain engineering are captured as *aspects*, thus enabling components to be reused on a variety of execution scenarios through the application of *aspect programs*. The design artifacts proposed by ADESD facilitates the

separation of concerns within embedded system components making hybrid implementations possible. This was initially explored by the idea of *Hybrid Components* [27], in which a common architecture for the implementation and communication between components living in hardware and/or software was developed. This architecture defines interfaces that, when implemented, allow components to communicate independently of whether they are deployed in software or in hardware.

In this work we aim to show how ADESD's aspect separation mechanisms can be used to yield components susceptible to both hardware and software synthesis. ADESD proposes the use of constructs called *Scenario adapters* [28] to perform aspect weaving. 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. Figure 1 shows how this is achieved. The *Scenario* class represents the execution scenario and incorporates, via aggregation, all aspect programs that are needed to characterize it. The adaptation of the component to the scenario is performed by the *Scenario_Adapter* class via inheritance implemented using static metaprogramming.

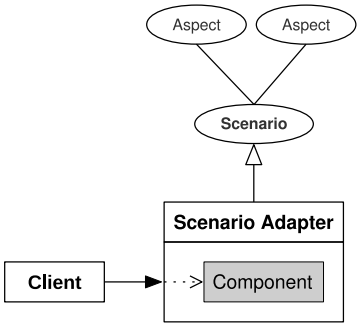

Figure 1. Component adaptation using a scenario adapter.

In the next sections we demonstrate how characteristics specific of a *hardware scenario* or a *software scenario* can be isolated from the core behavior and the interface of a component. By following this approach, the same component description can be tailored by the scenario adapter in order to feed either a software compilation flow or a hardware high-level synthesis flow.

4 Unified Hardware/Software Design

HLS tools allow for hardware synthesis from algorithms described in languages such as C++. When aiming at this goal, the implementation of the algorithm itself is usually very similar to what is seen in software. However, at component level, several characteristics arise which distinguish descriptions aiming at hardware and software. In the following sections we first discuss these differences. Then, we present our strategy for the separation of hardware and software concerns and how we achieved efficient and flexible unified C++ descriptions.

4.1 Differences between hardware and software

Table 1 summarizes the most common components communication patterns. In the software domain, components may be objects which communicate using method invocation (considering an OOP-based approach), while in the hardware domain, components communicate using input/output signals and specific handshaking protocols. For communication through different domains, the software must provide appropriate *hardware abstraction layers* (HAL) and *interrupt service routines* (ISR), while the hardware must be aware that it is requesting a software operation.

Table 1
Usual component communication patterns. The *Caller* requests operations from the *Callee*.

Direction	Type of communication	
	Caller	Callee
SW→HW	HAL sends commands to the HW using a communication infrastructure (e.g. a bus or a NoC).	Communication interface receives commands and triggers the operations.
HW→SW	HW interrupts SW and waits for the operation to finish. It may transfer data to/from the main memory (e.g. DMA).	An ISR calls the requested operation and signalizes HW when finished.
SW→SW	Function call interface	
HW→HW	Signals/Handshaking	

In some system-level approaches, such as TLM, components communicate by reading and writing data from/to *channels*. Latter in the design process, these channels can be mapped to buses, point-to-point signals, or function calls, whether the communicating components are in hardware or in software. This strategy provides a clear separation between communication and behavior, but it is still too hardware-oriented since it basically provides higher-level versions of RTL signals. Figure 2 highlights this fact by comparing a TLM and a more software-oriented OO model of the same system. The OOP model is more expressive and clear in the way components interactions are defined. A component can be either implemented as a global object (*C1* and *C3*) or as a part of another object (*C2*).

However, the structure of TLM-like models leads to a more natural mapping to final physical implementations. In OOP the original structure will be "disassembled" if different objects in the same class hierarchy represent components that are to be implemented in different domains. For example, *C2* is "inside" *C1* in the OO model in Figure 2, but, in the final implementation, *C1* could be implemented as a hardware component while *C2* could run as software in a processor. Since we focus on an OOP-based methodology, in this paper we show how these different mappings can be made transparent

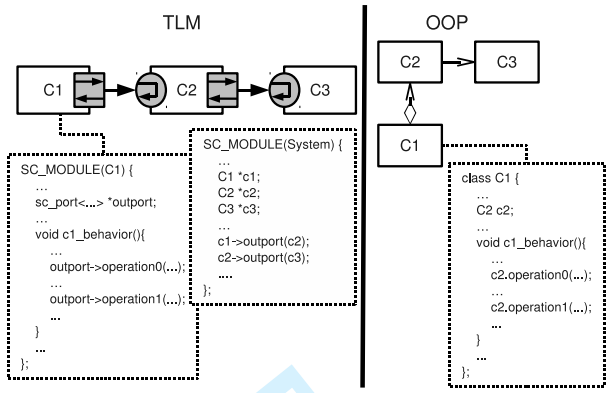

Figure 2. Communication between components in transaction level (left) and object-oriented (right) models.

to the designer. This is explained in section 4.2.3.

Another important characteristic that distinguishes hardware from software is resource allocation. The hardware is frozen and common software features, such as dynamic resource allocation, are not easily available. Therefore, in code suitable for hardware synthesis, all data structures must reside in statically allocated memory. Some works focus on this problem by relying on dynamic reconfiguration technologies of FPGAs to support hardware components instantiation at run-time[29]. However, this is not the focus of our current work. We aim at providing more general guidelines for describing components.

4.2 Defining C++ unified descriptions

A unified description of components can be obtained from implementations that are largely independent from hardware and software scenario details. This can be achieved through careful domain engineering and system design. An initial work [30] has already shown how the ADESD methodology was used to provide a C++ description of a resource scheduler suitable for high-level synthesis. In the next sections we show how one can further leverage on ADESD's artifacts to provide C++ unified descriptions and efficient hardware/software aspects separation.

4.2.1 Component implementation

C++ code unified and suitable for automatic implementation in both hardware and software must follow a careful design process so it will not contain characteristics specific of hardware or software. To illustrate how ADESD yields such components, we describe in more details one of our case studies: the design and unified implementation of the EPOS's thread scheduler. The *Embedded Parallel Operating System* (EPOS) [31] is a case study on which the design artifacts proposed by ADESD were implemented and validated. The OS is implemented in C++ and leverages on *generative programming*[7] techniques such as *static metaprogramming* in order to achieve high reusability with low overhead. EPOS's domain engineering simplified our work by providing a good *separation of concerns* around the scheduler. Figure 3 shows a simplified class diagram of the scheduling-related classes in EPOS.

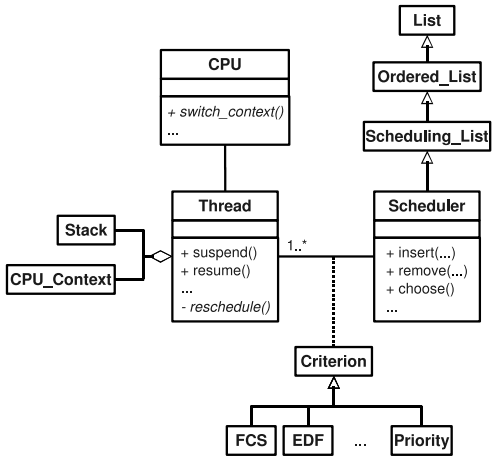

Figure 3. Simplified class diagram of scheduling-related classes in EPOS.

The *Scheduler* class is responsible only for keeping ordered queues, with the ordering defined by the scheduling criterion. An object of the *Scheduler* class also allows its callers to insert and remove resources from the queues, as well as to get the current and next owners of the resource. In our case study, we implemented a scheduler for threads. However, the scheduler code is largely independent of the type of resource being scheduled (in fact, *Scheduler* is a *parametric class*). Furthermore, concerns such as timing interrupt generation and context switching are handled separately by the OS.

Among all classes in Figure 3, we've adapted the original implementation of the *Scheduler* class (and also its base classes), so that it can now serve as input for both hardware and software implementation flows. These adaptations were necessary not due to specific hardware or software characteristics, but due to some limitations of HLS tools. The main limitation is related to *pointers*. Pointers have no intrinsic meaning in hardware. They are mapped by HLS tools to indices of the storage structures to which they point. Thus, *no null or otherwise invalid pointers* are allowed in the source code. However, the original scheduler implementation used null pointers to report failures. To avoid this, we've changed the code to utilize *option types*. An option type is a container for a generic value, and has an internal state which represents the presence or absence of this value. We implemented an option type in the C++ class template *SafePointer<T>*, which has the following constructors:

```
template<typename T> class SafePointer {
...
  SafePointer(): _exists(false), _thing(T()){}
  SafePointer(T obj): _exists(true), _thing(obj){}
...
};
```

One constructor represents the absence of a value in the container while the other represents its presence. By replacing all occurrences of simple pointers (T*) in the scheduler code with *SafePointer<T*>* values, we completely avoided passing invalid pointers around.

HLS tools also limit the use of C++ features that rely on dynamic structures in software (e.g. heaps, stacks, virtual method tables), such as *new* and *delete* operators, recursion, and dynamic polymorphism. In section 4.2.2 we show how resource allocation can be implemented as a separated *aspect*, however recursion and polymorphism can be implemented using static metaprogramming techniques, yielding code supported by HLS tools. Static polymorphism can be implemented as shown below:

```
template <typename derived_class> class Base {
  void operation() {
    static_cast<derived_class*>(this)->operation();
  }
};
class Derived : public Base<Derived> {
  void operation();
};
```

Classes can derive from template instantiations of the base class using themselves as template. This is also known as *Curiously Recurring Template Pattern* (CRTP) [32], and allows the static resolution of calls to virtual methods of the base class.

Other aspects concerning hardware generation using HLS are related to the synthesis process. The same high-level algorithm can span several different hardware implementations. For instance, loops can have each iteration executed in a clock cycle, or can be fully unrolled in order to increase throughput at the cost of additional silicon area. This kind of synthesis decision is usually taken based on directives which are provided separately from the algorithm descriptions. The definition and fine tuning of these directives is part of the design space exploration process and is not in the scope of this work.

4.2.2 HW/SW aspects encapsulation

Throughout the domain decompositions of our case studies we have identified two aspects that must be handled differently whether the component is in hardware or in software: *storage allocation* and *method call interface*. Figure 4 shows how we have used scenario adapters to isolate such aspects.

The *HW Scenario* is composed by the aspects *Static Alloc* and *Dispatch*, whom are responsible, respectively, for storage allocation and method call dispatch. The former is a storage allocator used to deal with the absence of dynamic memory allocation in hardware. Handling memory allocation externally is only possible because our components implement solely their *core behavior*. For example, the *List* class (Figure 3) is a linked list, but it is not responsible for the allocation of space for links and the objects it stores. It implements only the list algorithms and deals with references to such elements. Therefore, all components operations go through the

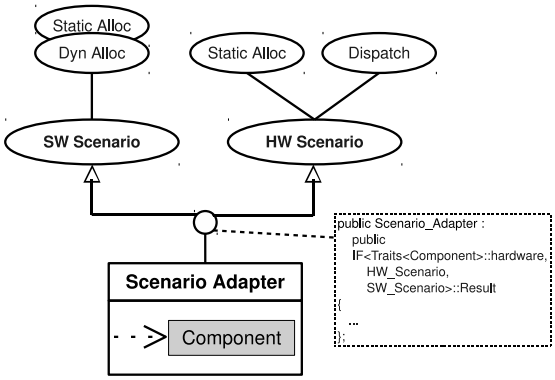

Figure 4. Software and hardware scenarios' aspects

allocator which reserves and releases storage space on demand. The example below shows how the *insert* and *remove* methods of the list are implemented using this approach:

```
Link insert(Object obj) {
    Link link = Scenario::allocate(obj);
    Base::insert(link);
    return link;
}
Object remove(Link link) {
    Object obj = Scenario::get(link);
    Base::remove(link);
    Scenario::free(link);
    return obj;
}
```

In hardware implementations, *Scenario::allocate(obj)*, *Scenario::get(link)*, and *Scenario::free(link)* map to operations implemented in the *Static Alloc* aspect. In this scenario, the number of storage slots for links and objects is defined at synthesis-time and allocation requests are just mapped to a free slot. In a software implementation, dynamic memory allocation is available, thus, storage allocation can be handled by either the *Static Alloc* or the *Dynamic Alloc* aspect, as shown in figure 4.

The *Dispatch* aspect is used, in *HW Scenario*, to define an entry point for the component so it will be compliant with HLS tools requirements. For the tool used in our case studies (Calypto's CatapultC [1]), the top-level interface of the resulting hardware block (port directions and sizes) is inferred from a *single function signature*. This function is defined by *Dispatch* and receives a *method id* as its first parameter, interprets its value, performs the necessary type conversions and calls the appropriate method of the component. The value returned by the called method is also inspected, converted if necessary and assigned to one of the dispatcher output parameters. A dispatch mechanism is not necessary in the software scenario since operations are requested using direct method calls.

Finally, it is worth mentioning that aspects, scenario, and adapters are implemented using static metaprogramming and are highly generic. The scenario to which the component is adapted is, in fact, defined using metaprogramming, as shown in figure 4. Special template classes called *Traits* are used to define which

characteristics of each component is activated. The code sample below shows an example of a *Trait* class:

```
template <> struct Traits<Component> {
    static const bool hardware = true;
};
```

It defines that the component *Component* has a *hardware* characteristic that is used to define which domain the component will be on (hardware or software). In figure 4, this characteristic is used to conditionally modify the scenario adapter's base scenario. This decision is statically determined using a metaprogram. The implementation of *IF* metaprogram 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; };
```

4.2.3 Wrapping communication

Issues related to communication cannot be completely handled using only the mechanisms described so far. For example, in the OO model in Figure 2, the *C2* is an attribute of *C1*, thus *C2* will be compiled/synthesized together with *C1*. However, if the designer intends to have *C1* and *C2* as two independent components implemented in the different domains, then an additional mechanism is necessary.

A way to overcome this issue is to use to use concepts from distributed object platforms [26]. Figure 5 illustrates possible interactions between components *C1* and *C2*. The callee is represented in the domain of the caller by a *proxy*. When an operation is invoked on the components' proxy, the arguments supplied are marshaled in a request message and sent through a *communication channel* to the actual component. In the *HW->SW* interaction, an *agent* receives requests, unpacks the arguments and performs local method invocations. The whole process is then repeated in the opposite direction, producing reply messages that carry eventual return arguments back to the caller. An *agent* is not required by the *SW->HW* interaction since the *dispatcher aspect* can already play its role.

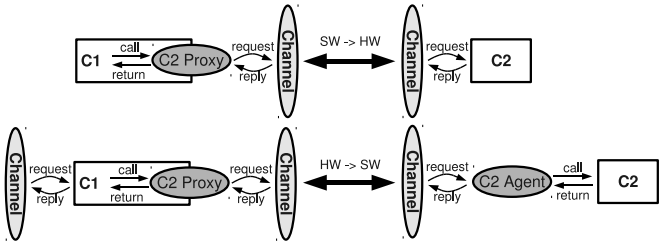

Figure 5. Communication between components in different domains. The leftmost components request operations from the rightmost ones (the scenario adapters are omitted for simplicity).

Another issue is how to make these mechanisms transparent in the components' descriptions. An efficient solution is to use template metaprograms to replace the definition of a component by its proxy when necessary. For example, in the software domain, *C2* can be defined as an empty class that inherits from its actual implementation depending on the configuration defined by the component's *Traits*:

```
class C2 :
  public IF<Traits<C2>::hardware,
            SW_Proxy<C2>, Scenario_Adapter<C2> >::Result {};
```

There are still more implementation issues that are still to be considered. The mechanisms proposed above are only general guidelines that can span several specific implementations. The implementation of *channels*, *proxies* and *agents* can be realized in several different ways (e.g. specific using buses, DMA, *Network-on-Chips* (NoCs)). In section 5 we describe our case studies and the chosen architecture to implement the mechanisms above.

4.3 Implementation flow summary

Figure 6 summarizes our implementation flow. The first steps show the artifacts of our proposed approach. Aspects, scenarios, and the unified implementation of components made up the inputs for the design flows. Proxies and agents can be automatically generated from the components' interface by a tool (EPOS's syntactical analyzer [33] can be used to obtain the operation signatures of all components). The final steps comprises the final system generation. Once the hardware/software partitioning is defined (by the components' Traits), the adapted components are fed to either the hardware or the software generation flows.

In the software flow, components are compiled along with the RTOS. In our case study we use EPOS. Apart of basic RTOS features, EPOS is also responsible for the implementation of the run-time support required by the mechanisms mentioned in section 4.2.3. For example, a software component that is in the scope of a hardware component is instantiated by this run-time support. When the hardware component requests operations through the proxy, the software is interrupted and the run-time support handles the communication between the stub and the component in software.

In the hardware flow, an HLS tool is used to generate RTL descriptions. These descriptions are bound with the hardware platform library and fed to the RTL synthesis tool. The platform library contains the components required to run the software (softcore processor, memories, and peripherals) and the communication infrastructure that allows application's components to communicate with each other. Details of the features available in both EPOS and the platform library are given in the description of our case study in the next section.

It is worth pointing out that this work focuses only on general implementation guidelines and not a whole design methodology. Issues such as design space exploration and design verification are not in the scope of this work.

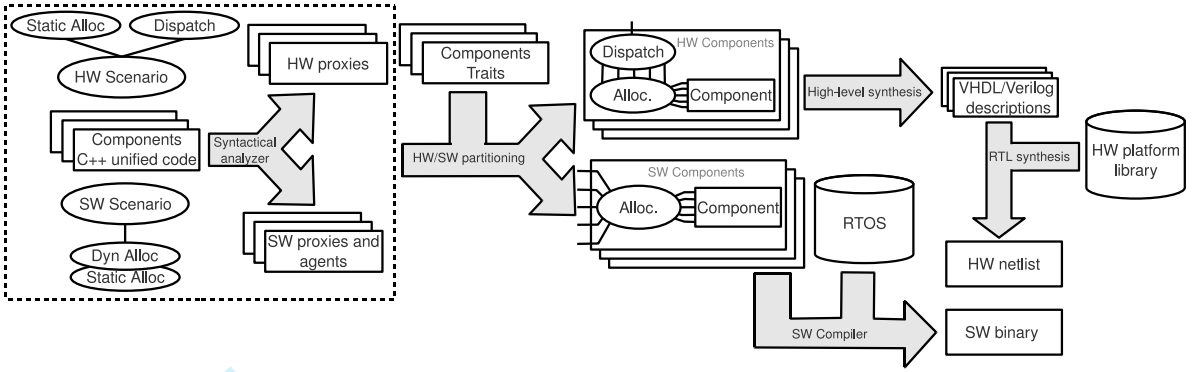

Figure 6. Summary of the implementation flow

5 Case Study

In order to evaluate our approach, we have analyzed a PABX application and designed some of its basic building blocks using the proposed implementation guidelines. A high-level diagram of the PABX system is shown in Figure 7. The main component is 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 supports the transmission of voice data through an Ethernet network.

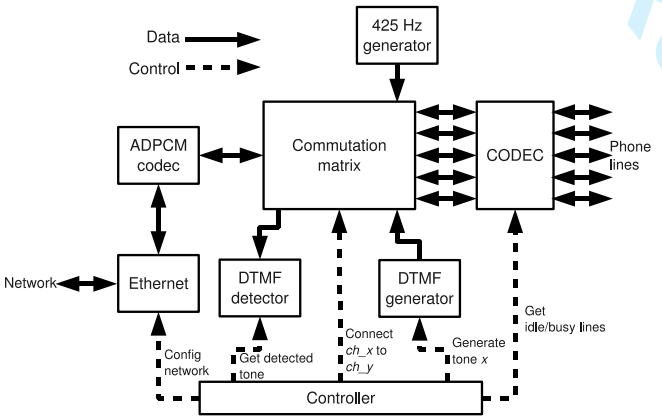

Figure 7. PABX system diagram

The components we have selected to be reimplemented using unified C++ are described in more detail below:

EPOS's scheduler: the use of the scheduler described in section 4.2.1 as a case study is motivated by its complex behavior. A scheduler may perform operations both synchronously (upon request by another component) or asynchronously (by preempting the execution of another component). Furthermore, a hardware-implemented scheduler has less jitter and provides better support for real-time applications [34]. Figure 8 extends figure 3 with the main operations implemented by the scheduler. For our thread scheduler, the component is instantiated using class *Thread* as the template parameter.

ADPCM codec: an IMA ADPCM encoder/decoder [35] that is used to reduce the traffic of data transmitted through the network. It performs data compression using an *adaptative differential pulse-code modulation* (ADPCM) algorithm to convert 16-bit samples to 4-bit samples. Figure 9 shows the structure of the codec. The encoding and decoding algorithms are independent and implemented in different classes. Both algorithms, however, share the same lookup tables which are defined in a common class. The *ADPCM_Codec* class only encloses both implementations as a single codec component.

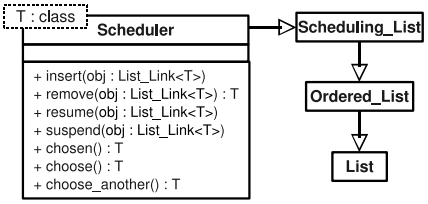

Figure 8. Scheduler component definition

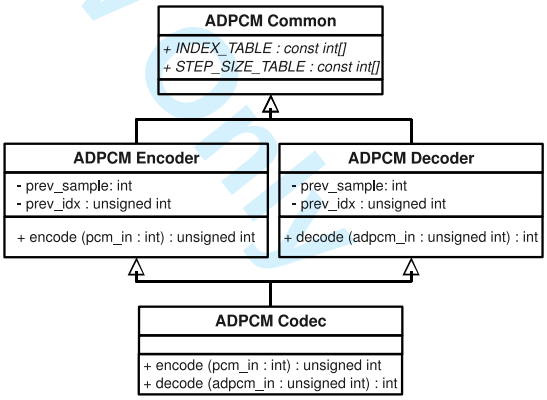

Figure 9. ADPCM codec component definition

DTMF detector: the *Dual-Tone Multi-Frequency* (DTMF) detector is one of the basic building blocks of any PABX system. The DTMF detector receives signals sampled from the phone lines connected to the central and detects DTMF tones. Figure 10 shows its definition. The entry point of the original C implementation was a single function that received a

pointer to a frame of samples and returned the detected tone. The redesigned unified implementation defines two public operations: *add_sample* updates a sample of the current frame; and *do_dtmf* implements the pseudocode in figure 10 to perform the detection. For each tone, it uses the *goertzel algorithm* to check if the sample frame contains the frequency components of a tone, then uses lookup tables to analyze the results and return the ASCII character representing the tone.

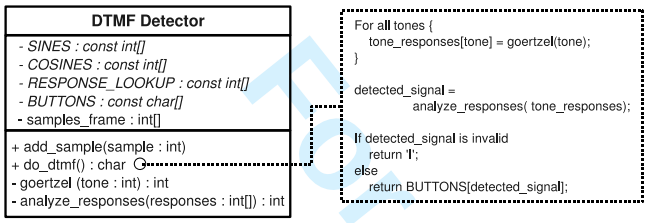

Figure 10. DTMF detector component definition

Figure 11 shows the final block diagram of the PABX SoC. It is a NoC-based SoC and uses the *RTSNoC*[36] as the core communication infrastructure. Each hardware component is implemented as a single node connected to the NoC. Software components run in a *processing node* that consists of a MIPS32 core and memories. The SoC also contains an *IO peripheral node*. The components selected as case studies are highlighted and appear in as hardware and software, since they can move between both domains depending on the final hardware/software partitioning.

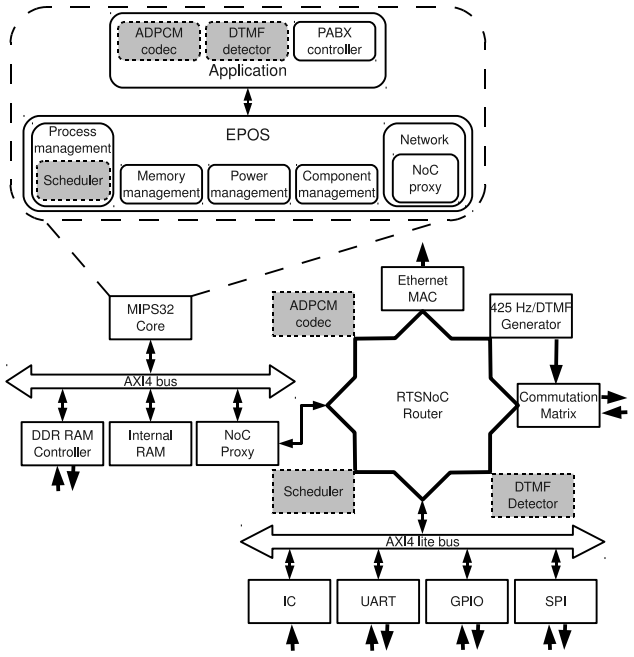

Figure 11. PABX SoC block diagram

The upper part of figure 11 shows the structure of the software domain. EPOS's *Component Manager* implements the runtime support mentioned in section 4.3. Operations request from software to hardware are marshaled and sent through the *NoC proxy*. Requests in the opposite direction are captured by the *Component Manager* which forwards the operation to the target component in software (it implements the software agents described in section 4.2.3).

5.1 Results

In order to demonstrate that unified implementations can be compared to dedicated ones in terms of efficiency, we compare software scenario-adapted components against the original C implementations (C++ for the scheduler), and hardware scenario-adapted components against components manually tailored for high-level synthesis.

Table 2 and Figures 12–13 compare the original software-only C (ADPCM and DTMF) and C++ (Scheduler) with the software scenario-adapted C++. The footprints were obtained from the object files generated after compiling each component in isolation, while the execution times were measured from the running application using a timestamp counter. Everything was compiled with *gcc 4.0.2* targeting the MIPS32 ISA and using *level 2* optimizations. Figure 12 shows an average increase of about 4.9% in the total memory footprint. In the case of the scheduler, the overhead can be explained by the introduction of the *option types* and the use of a more generic mechanism for storage allocation (the *Static/Dyn Alloc* aspect). For the remaining case studies, most of the overhead comes from additional code required to encapsulate the behavior into more reusable OOP classes with a clear method interface.

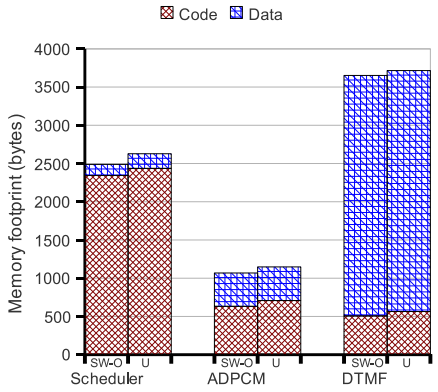

Figure 12. Memory footprint of software-only C++ *vs.* Unified C++ adapted to the software domain. The leftmost bars show results for software-only components.

Table 2 and Figure 13 show the execution time of each component operation and compare the average values. The execution time of the Scheduler and the ADPCM codec is about 2.5% higher in the unified implementation. For the DTMF detector, the difference increases to 5%. The original DTMF detector requires a single call to *do_dtmf* to analyze a frame of samples,

Table 2
Execution time of software-only C++ *vs.* Unified C++ adapted to the software domain.

Component		Execution time (μs)	
		SW-only	Unified
Scheduler	insert	6.0	6.0
	remove	2.9	3.3
	suspend	3.0	3.1
	resume	6.0	6.0
	choose	8.4	8.5
ADPCM	encode	4.2	4.2
	decode	3.4	3.6
DTMF	do_dtmf	5878.3	6182.9

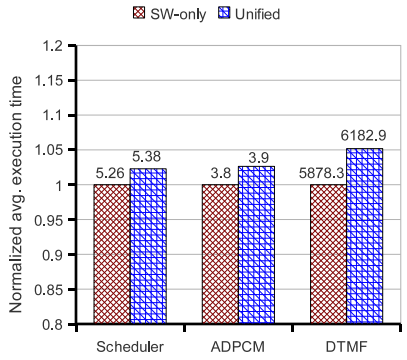

Figure 13. Normalized average execution times. Absolute values are shown above their respective bars.

while the refactored DTMF detector requires several calls to *add_sample* before performing the same task, which results in a more significant increase in the execution time.

Table 3 and Figures 14–15 compare the hardware generated from unified C++ against hardware-only C++. *Calypto's CatapultC* HLS tool was used to obtain RTL descriptions of the components. The descriptions were then synthesized using *Xilinx's ISE 13.4* targeting a *Virtex6 XC6VLX240T* FPGA. CatapultC and ISE were configured to minimize circuit area considering a target operating frequency of 100 MHz (the operating frequency of the SoC).

Table 3
FPGA resource utilization of hardware-only implementations *vs.* Unified implementations adapted to the hardware domain

Resource	Scheduler		ADPCM		DTMF	
	HW-only	Unified	HW-only	Unified	HW-only	Unified
6-input LUT	2119	2540	524	615	387	443
Flip-flops	1849	2766	208	368	331	431
36Kb BRAM	0	0	1	1	2	1
DSP slice	0	0	0	0	6	6

Table 3 shows the amount of FPGA resources required for each component and Figure 14 plots the *average*

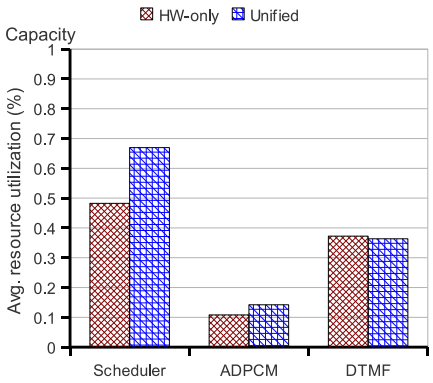

Figure 14. Average FPGA resource utilization

resource utilization. The *average resource utilization* is the arithmetic mean of the amount of each specific resource weighted by its total amount available. This resulting value estimates the total amount of FPGA area (in %) required and is used as an area comparison metric. The results vary significantly in each case study. The apparent smaller area of the unified DTMF detector is due to different mapping of resources between *RAM blocks* and *flip-flops*, which resulted in smaller *average resource utilization*. On the other hand, the unified Scheduler and DTMF require about 39% and 27% more area, respectively. Since the implementations of the HW-only and unified ADPCMs are very similar, we conclude that, in this case, most of the extra area comes from the *Dispatch* aspect which implements the *agent* (section 4.2.3) required for transparent communication between hardware and software. The agent must implement a generic mechanism for parsing and issuing operation requests, while a hardware-only description can focus on a more specific and optimized interface. This overhead can be considered significant and increases with the number of supported operations. Nevertheless, this overhead is expected to be significantly reduced in future implementations of the agent.

Figure 15a shows the maximum operating frequency of each component. All implementations achieved similar clock frequencies and met the 100 MHz constraint. Figure 15b shows the maximum number of clock cycles required to perform an operation. For the unified implementations of the Scheduler and the ADPCM, the *agents'* overhead is proportional to the number of arguments in the operation (2 for *Scheduler::insert* and 1 for *ADPCM_Codec::encode*). The high absolute overhead of the unified DTMF detector comes from the several invocations of *DTMF_Detector::add_sample* required to fill its internal buffer. This operation is implemented in a stream-like fashion in the HW-only detector.

To conclude our analysis, we have evaluated the total overhead of communication infrastructure required to handle transparent hardware/software communication. EPOS's internal components are implemented using static metaprogramming to achieve high reusability with

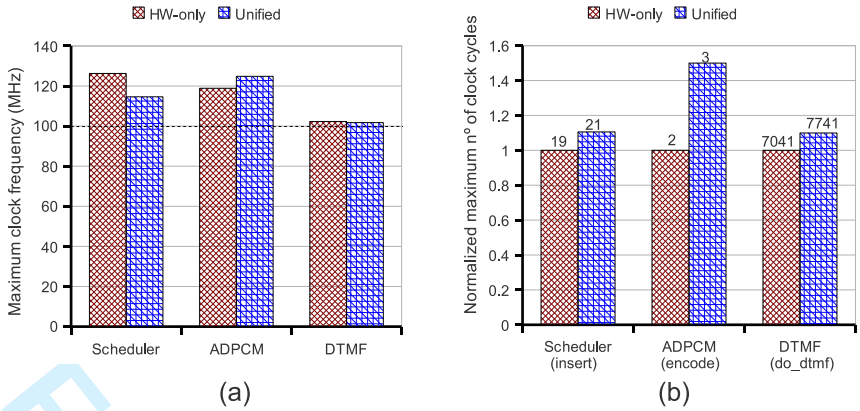

Figure 15. Performance of hardware-only C++ *vs.* unified C++ adapted to the hardware domain. In figure (b) values are normalized to compare all components in the same scale.

Table 4
Hardware footprint of the remaining components

	Cmm. matrix	425Hz/DTMF gen.	Eth MAC	Proc. node	IO node	RTSNoC
6-input LUTs	93	519	3424	7775	537	2256
Flip-flops	108	544	4967	6834	409	689
36 Kb Block RAM	0	0	0	2	0	0
Avg. rsc. utilization	0.03	0.13	1.11	1.90	0.11	0.30

low overhead and its final footprint depends on the application configuration. Therefore, it does not make sense to evaluate certain components in isolation (e.g. the *Component Manager*—Figure 11, which implements the runtime support described in section 4.3). In order to provide such evaluation, Table 5 thus shows the total footprint considering the following different partitionings of our PABX SoC: all the case studies are in software; all the case studies are in hardware; and a hybrid partitioning in which only the scheduler is in software. The values shown in parentheses are the total footprints subtracted by the footprints of all case studies in the respective domain. For instance, the software footprint *()'s* value of a specific partitioning is the total software footprint subtracted by the ones of all case studies implemented as software in that partitioning. This value allows us to evaluate the overhead added by the component management mechanisms. As reference, the footprints of the remaining hardware components are also provided in Table 4.

As can be seen in table 5, moving all cases to hardware reduced the total footprint. By comparing the *()'s* values, we can see that, in the *All HW* partitioning, the introduction of the component management mechanism added an overhead of 4278 bytes. By moving the Scheduler back to software in the *Hybrid* partitioning, the aforementioned overhead is reduced to 4170 bytes. Considering the number of operations implemented in each stub, we estimate an initial overhead of about 4 Kbytes plus 30 bytes for each operation implemented in a stub. We expect to significantly reduce this overhead in future implementations, however, the current results are acceptable if compared with similar infrastructures [37]. On the hardware side, the calculated area overhead when all cases are in hardware represents only 1.3% of the total circuit area and negligible when represented in terms of the total amount of resources available in the target device (only 0.05%).

Table 5
SoC area footprint. In the hybrid partitioning, the Scheduler is in software while the ADPCM codec and the DTMF detector are in hardware.

	Memory (code+data)	FPGA Avg. rsc. util.
All SW	19553 b (12201 b)	3.70% (3.70%)
All HW	16479 b (16479 b)	5.05% (3.75%)
Hybrid	19093 b (16605 b)	4.24% (3.74%)

6 Conclusion

In this paper we have explored a methodology based on AOP and OOP concepts in order to produce unified descriptions of hardware and software components. We have shown that components designed following the principles presented in this work are susceptible to both software and hardware generation using standard compilers and HLS tools. This is possible through the isolation of specific hardware and software characteristics (resource allocation and communication interface) into aspect programs which are weaved with the unified descriptions only during the final stages of the design

process. Furthermore, our mechanisms are implemented using only standard C++ features, thus facilitating compatibility with different C++/C-based HLS tools.

Finally, we have demonstrated our methods by redesigning some functional blocks of a PABX system. The resulting components confirmed that, at an acceptable cost in area and performance, we can use C++ as a unified language to implement both hardware and software in an straightforward way, thus reducing the costs of design cycles and time-to-market, and contributing to the progress of embedded system design towards system-level methodologies.

Acknowledgments

The authors would like to thank João Paulo Pizani Flor and Marcelo Daniel Berejuck for providing the original implementations of the some of the case studies used in this work. This work is also partially supported by the *Coordination for Improvement of Higher Level Personnel* (CAPES), under grants RH-TVD 006/2008 and 240/2008.

References

[1] Calypto Design Systems, "CatapultC Synthesis," 2011, http://www.calypto.com/.
[2] Synopsys, "Synphony C Compiler," 2011, http://www.synopsys.com.
[3] Cadence Design Systems, "C-to-Silicon Compiler," 2011, http://www.cadence.com.
[4] Forte Design Systems, "Cynthesizer," 2011, http://www.forteds.com.
[5] Xilinx, "AutoESL High-Level Synthesis," 2012, http://www.xilinx.com/tools/autoesl.htm.
[6] F. V. Polpeta and A. A. Fröhlich, "On the Automatic Generation of SoC-based Embedded Systems," in *Proc. of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy, Sep 2005.
[7] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. New York, USA: ACM Press/Addison-Wesley Publishing Co., 2000.
[8] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "PeaCE: A hardware-software codesign environment for multimedia embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, pp. 24:1–24:25, May 2008.
[9] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming Heterogeneity - The Ptolemy Approach," in *Proc. of the IEEE*, 2003, pp. 127–144.
[10] M.-A. Dziri, W. Cesario, F. Wagner, and A. Jerraya, "Unified component integration flow for multi-processor SoC design and validation," in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, 2004, pp. 1132–1137 Vol.2.
[11] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: an integrated electronic system design environment," *Computer*, vol. 36, pp. 45–52, April 2003.
[12] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu, "A Next-Generation Design Framework for Platform-based Design," in *Proc. of the Design & Verification Conference & Exhibition*, February 2007.
[13] A. Sangiovanni-Vincentelli and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," *IEEE Design & Test of Computers*, vol. 18, pp. 23–33, 2001.
[14] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473 –491, april 2011.
[15] A. Schallenberg, W. Nebel, A. Herrholz, P. A. Hartmann, and F. Oppenheimer, "OSSS+R: a framework for application level modelling and synthesis of reconfigurable systems," in *Proc. of the Conference on Design, Automation and Test in Europe*, Nice, France, 2009, pp. 970–975.
[16] K. Grüttner, F. Oppenheimer, W. Nebel, F. Colas-Bigey, and A.-M. Fouilliart, "SystemC-based modelling, seamless refinement, and synthesis of a JPEG 2000 decoder," in *Proc. of the conference on Design, automation and test in Europe*, Munich, Germany, 2008, pp. 128–133.
[17] F. Mischkalla, D. He, and W. Mueller, "Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems," in *Proc. of the Conference on Design, Automation and Test in Europe*, Dresden, Germany, 2010, pp. 1201–1206.
[18] OMG, *OMG Systems Modeling Language (SysML)*, 2010. [Online]. Available: http://www.sysml.org
[19] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, pp. 1:1–1:23, January 2009.
[20] J. Falk, C. Haubelt, and J. Teich, "Efficient Representation and Simulation of Model-Based Designs in SystemC," in *Proc. of the Forum on Design Languages*, 2006, pp. 129–134.
[21] D. Rainer, G. Andreas, P. Junyu, S. Dongwan, C. Lukai, Y. Haobo, A. Samar, D. Daniel *et al.*, "System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design," *EURASIP Journal on Embedded Systems*, vol. 2008, 2008.
[22] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proc. of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Newport Beach, USA, 2003, pp. 19–24.
[23] E. Anderson, W. Peck, J. Stevens, J. Agron, F. Baijot, S. Warn, and D. Andrews, "Supporting High Level Language Semantics within Hardware Resident Threads," in *Proc. of the International Conference on Field Programmable Logic and Applications.*, Aug. 2007, pp. 98 –103.
[24] E. Lubbers and M. Platzner, "A portable abstraction layer for hardware threads," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, sept. 2008, pp. 17 –22.
[25] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 14:1–14:28, January 2008.
[26] F. Rincón, J. Barba, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, and J. C. López, "Transparent IP Cores Integration Based on the Distributed Object Paradigm," in *Intelligent Technical Systems*, ser. Lecture Notes in Electrical Engineering, 2009, vol. 38, pp. 131–144.
[27] H. Marcondes and A. A. Fröhlich, "A Hybrid Hardware and Software Component Architecture for Embedded System Design," in *Proc. of the International Embedded Systems Symposium*, Langenargen, Germany, 2009, pp. 259–270.
[28] T. R. Mück, M. Gernoth, W. Schröder-Preikschat, and A. A. Fröhlich, "Implementing OS Components in Hardware using AOP," *SIGOPS Operating Systems Review*, vol. 46, no. 1, pp. 64–72, 2012.
[29] N. Abel, "Design and Implementation of an Object-Oriented Framework for Dynamic Partial Reconfiguration," in *Proc. of the International Conference on Field Programmable Logic and Applications*, September 2010, pp. 240 –243.
[30] J. P. P. Flor, T. R. Mück, and A. A. Fröhlich, "High-level Design and Synthesis of a Resource Scheduler," in *Proc. of the 18th IEEE International Conference on Electronics, Circuits, and Systems*, Beirut, Lebanon, 2011, pp. 736–739.
[31] The EPOS Project, "Embedded Parallel Operating System," 2011, http://epos.lisha.ufsc.br/.
[32] J. O. Coplien, "Curiously recurring template patterns," *C++ Rep.*, vol. 7, pp. 24–27, February 1995.
[33] A. Schulter, R. Cancian, M. R. Stemmer, and A. A. M. Fröhlich, "A Tool for Supporting and Automating the Development of Component-based Embedded Systems," *Journal of Object Technology*, vol. 6, no. 9, pp. 399–416, Oct 2007.
[34] H. Marcondes, R. Cancian, M. Stemmer, and A. A. Fröhlich, "On the Design of Flexible Real-Time Schedulers for Embedded Systems," in *Proc. of the 2009 International Conference on Computational Science and Engineering - Volume 02*, Washington, USA, 2009, pp. 382–387.

[35] Interactive Multimedia Association (IMA), *Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems*, 1992. [Online]. Available: http://www.cs.columbia.edu/~hgs/audio/dvi/IMA_ADPCM.pdf
[36] M. D. Berejuck, "Dynamic Reconfiguration Support for FPGA-based Real-time Systems," Federal University of Santa Catarina, Florianópolis, Brazil, Tech. Rep., 2011, PhD qualifying report.
[37] G. Gracioli and A. A. Fröhlich, "ELUS: A dynamic software reconfiguration infrastructure for embedded systems," in *Proc. of the IEEE 17th International Conference on Telecommunications*, april 2010, pp. 981 –988.