Unified Design of Hardware and Software Components

Journal:	*Transactions on Computers*
Manuscript ID:	TC-2012-06-0397.R1
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, C.0.e System architectures, integration and modeling < C.0 General < C Computer Systems Organization, D.2.10.g Object-oriented design methods < D.2.10 Design < D.2 Software Engineering < D Software/Software Engineering

Tiago Rogério Mück and
Prof. Dr. Antônio Augusto Fröhlich
Federal University of Santa Catarina
Florianópolis, 88040-000, Brazil
Phone: +55 (48) 3721-9516
E-Mail: {tiago,guto}@lisha.ufsc.br

November 20, 2012

IEEE Transactions on Computers
Editor-in-Chief
Prof. Dr. Albert Y. Zomaya

Dear Prof. Zomaya,

A revised version of the manuscript, "Unified Design of Hardware and Software Components", initially submitted to the IEEE Transactions on Computers on July 2012, has just been uploaded through the manuscript submission site.

First of all, we would like to thank the reviewers for the detailed discussion. In summary, the main changes carried out in this resubmission were:

- In section 3, we added an explanation about basic OOP concepts and extended the description of ADESD and AOP;
- Minor changes in section 4.1 to make the purpose of Figure 2 clearer;
- To clarify the points highlighted by the reviewers and improve the paper organization, major changes were performed in the remaining of section 4:
  - We extended the original section *4.2.1 Component implementation* and merged its contents in to the top-level section *4.2 Defining C++ unified descriptions*;
  - Concerns related to the limitations of high-level synthesis were moved to a new section *4.2.1 Synthesis considerations*;
  - We added a paragraph about bit-accurate data types in the new section 4.2.1;
  - The original subsection 4.2.2 became subsection 4.3 and was extended to provide a more detailed explanation about our approach to apply hardware/software aspects and to consider alternative solutions;
  - Former sections *4.2.3 Wrapping communication* and *4.3 Implementation flow summary* moved to a new section *5 Deployment of unified components*;
  - We added a new section *4.4 Summary and discussion* that summarizes our approach and discuss other methods for describing hardware and software in a common language;
- We extended section 5 with more details about the implementation of proxies and agents;
- Minor changes in section 6 (former section 5):
  - Details of the base SoC platform moved to section 5;
  - Error bars added to measured execution times;
  - Improved the readability of Table 5;
  - Minor changes to reduce the section size

Please find below a more detailed description of how we have addressed the issues pointed out by each reviewer.

Reviewer #1

1. Reviewer's comment: *In my opinion, the main weakness of the paper lies in the presentation of proposed solution in section 4. Instead of giving general guidelines, the authors only present sample code snippets, which provide solutions for their running example.*

   We have carefully analyzed the way we present our approach, and we agree that the most general

guidelines were indeed not very clear. In summary, these guidelines consists in the following: 1) limiting component interaction to method calls allow the creation of a generic external mechanism; 2) components that require resource allocation must be designed in order to deal with links to the expected resources, thus allowing such allocation to be performed externally using the most suitable approach for hardware or software; and 3) the C++ in the unified implementation must be synthesizable. We have reorganized and extended section 4 in order to make the considerations above clearer.

The latter guideline does not affect the actual design of the components, but consists mostly of coding guidelines that must be followed due to current limitations of high-level synthesis. Such guidelines were all moved to a new section *4.2.1 Synthesis considerations*.

The former ones must be considered during the domain decomposition process; otherwise the incorporation of hardware/software characteristics using the proposed aspect weaving approach would not be possible. Since explaining the OOP decomposition process itself is not in the scope of this paper, we have showed the *Scheduler* as a case that satisfies the considerations mentioned above. In order to make clearer how this is achieved, we have extended the description of the Scheduler in section *4.2 Defining C++ unified descriptions*. Also, to better demonstrate how hardware/software characteristics can then be applied in a systematic way using the proposed scenario adapters, we have reorganized and extended section *4.3 HW/SW aspects encapsulation*: section 4.3.1 first presents the hardware/software aspects in details; section 4.3.2 then shows their aggregation to build a scenario; and section 4.3.3 describes the scenario adapter definition. We have also replaced the previous scenario diagram of Figure 4 by a more detailed one which shows the inheritance/template specialization structure more clearly.

We have also created a new section *4.4 Summary and discussion*. This section summarizes our strategy and highlights the points mentioned above. It also includes a discussion suggested in the comments 2 and 4 from reviewer #3.

2. Reviewer's comment: *Often, the code snippets are even not discussed at all, which makes it difficult to even comprehend this particular solution. Especially, I do have problems understanding the Traits example on page six in the right column.*

We thank the reviewer for pointing out these issues. We have improved the explanation of all code snippets. The provided explanations, however, assumes prior knowledge about object-oriented constructs in C++ and templates. Due to the page limit, we cannot review more general C++ concepts in the paper. Therefore, we have only added the following footnote (page 3, third paragraph) that provides some resources to which the reader may refer:

```
1. this paper assumes prior knowledge about OOP, UML and C++.
An overview of the relevant concepts is available at [31], [32], and
[33]. The reader may also refer to [34], [7], and [35] for an indepth
explanation.
```

The following additional references were added:

```
[31] "Wikipedia - Class diagram," 2012,
     http://en.wikipedia.org/wiki/Class_diagram.
[32] "Wikipedia - Object-oriented programming," 2012,
     http://en.wikipedia.org/wiki/Object-oriented_programming.
[33] The C++ Resources Network, "Templates," 2012,
     http://www.cplusplus.com/doc/tutorial/templates/.
[34] C. Larman, Applying UML And Patterns: An Introduction To Object-
     Oriented Analysis And Design And Iterative Development. Prentice
     Hall PTR, 2005.
[35] A. Alexandrescu, Modern C++ Design: Generic Programming and
     Design Patterns Applied, ser. C++ in-Depth Series.     Addison-
     Wesley, 2001.
```

Regarding the specific Traits example, we have extended the explanation of the trait concept in the second paragraph of section 4.3.2. Additional code examples are provided in sections 4.3.2 and 4.3.3. We have also added a reference to the original proposal of the trait concept, which provides additional examples:

```
[43] N. C. Myers, "Traits: a new and useful template technique," C++
     Report, June 1995. [Online]. Available: http://www.cantrip.org/
     traits.html
```

3. Reviewer's comment: *No alternative solutions are discussed. This would have been interesting especially for the static allocation and dispatching aspects. In particular, the proposed dispatching strategy is very*

specific for the CatapultC high-level synthesis. Nearly all other HLS tools do not use a single function signature for describing hardware components. They typically require to present hardware components as SystemC modules having signal ports or transaction sockets. In this case, the proposed solution could not be applied directly.

We agree with the reviewer in the sense that in our explanation we mentioned only one of the possible approaches; however, this is not a limitation for our proposal. Our current implementation of the dispatcher is compliant only with CatapultC since it is the tool used in our experimental evaluation. Nevertheless, the technical effort to define a top-level SystemC module for another HLS tool is about the same. To highlight the alternative solutions, we have added the following sentences in the last paragraph of section 4.3.1: 1) *In Calypto's CatapultC [1] and Xilinx's AutoESL [5], for instance, the top-level interface of the resulting hardware block (port directions and sizes) is inferred from a single function signature*; and 2) *Some tools require the definition of the entry point as a SystemC module with signal ports used to define the IO protocol. Our current implementation is compliant only with CatapultC, since it is the tool used in our experimental evaluation. Nevertheless, the Dispatch aspect can be specialized to support different entry-point requirements and different IO protocols (e.g. two-way handshaking, bus-based, etc). Upon system generation, the desired dispatcher can be selected using a* Trait.

We have also added more details to Figure 4, which makes clearer the specialization of the *Dispatch* aspect.

4. Reviewer's comment: *Moreover, why do the authors define their own allocation class while the standard template library is already providing a ready to use implementation? In summary, by only providing some examples, it is not clear what exact solution is proposed. Hence, the applicability of the solution and its limitations remain unclear.*

   In principle, we could have relied on STL; however, current STL implementations are not synthesizable. Nevertheless we agree that STL should be mentioned in the paper as a possible alternative. We have added the following sentence in the third paragraph of section 4.3.1: *A similar approach that uses external allocators for containers such as lists is provided by the C++ standard template library (STL) [42]. In principle, we could have relied on STL; however, current STL implementations are not synthesizable.*

   Also in section 4.3.1, we have extended the explanation of the List code example and added a code snippet showing part of the *Static Alloc* aspect implementation. We believe this will give the reader a clearer idea of the applicability of our allocators.

5. Reviewer's comment: *Although the paper addresses an important topic and the results clearly show the benefits of the proposed approach, the presentation of the key contributions is in my opinion too weak to accept the paper for publication. As only some illustrative examples are given without providing more general guidelines, the paper looks more like a case study presentation than a research paper.*

   In the answer for comment #1 we have described how we have addressed these issues. We would like to thank the reviewer for the constructive comments and we hope that the improvements to the manuscript will be sufficient to answer the main concerns raised by the reviewer.

Reviewer #2

1. Reviewer's comment: *Without reading the previous work of the authors it is very hard to get a deeper understanding of the technical concept and soundness of the approach. Section 3 should better aim on giving an introduction to the general concepts of aspects and aspect weaving. Currently section 3 points to previous work and gives only little high-level/abstract information.*

   We would like to thank the reviewer for pointing out this issue. We agree that a reader with little background on OOP, AOP and C++ may find it difficult to understand the ideas shown in the paper. In the revised manuscript, the first paragraph of section 3 presents some basic concepts of OOP. We also extended the description of AOP concepts and the ADESD methodology in the subsequent paragraphs. As mentioned by reviewer #3, an introduction to UML and C++ templates was also missing. However, due to the page limit, we cannot provide an in-depth explanation of these concepts, but we have added a footnote that provides additional references. Please refer to comment 2 from reviewer #1 for these changes.

2. Reviewer's comment: *In Section 4.1 the comparison of TLM-based communication and communication in object-oriented modes seems to be a little strange. Without giving further details on the methodology*

behind the presented approach (incl. different models for the application, execution platform and the mapping of application elements to component of the execution platform) this comparison is dangerous. Communication in object-oriented application models describe application specific communication, while TLM models describe how communication is realized in the execution platform through physical channels.

Our goal with Figure 2 is not to provide a direct comparison between OOP and TLM as methodologies for the same purpose. We believe the following sentence may induce the reader to this misunderstanding: *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.*

This sentence was removed and we have done minor changes in the second paragraph of section 4.2 to emphasize our main goal with Figure 2, which is to illustrate the problem that arises when an object-oriented approach is used to describe hardware/software components. As described in the paper: *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*. This also motivates the use of the proxy/agent mechanism described in section 5. Such problem does not exists in the TLM model, since, as highlighted by the reviewer, TLM is closer to the execution platform.

3. Reviewer's comment: *Section 4.2 aims at defining C++ unified description. This section appears to focus only on some specific issues like the use of pointers, static polymorphism, allocation, and dispatching. These issues are indeed important but the overall description of the unified description is missing. The presented techniques from template meta programming are interesting, but the description is not sufficient for understanding how these techniques are applied in a systematic way in an overall unified description.*

   We have described how we have addressed this issue in the answer to comments 1 and 2 from Reviewer #1.

4. Reviewer's comment: *Section 4.2.3 presents the idea of a Remote Method invocation with marshaling and unmarshaling services. It remains unclear how these techniques are supported by the presented methodology. Nothing about interrupt handling for software calls is mentioned.*

   We agree with the reviewer. The link between proxies/agents and their actual implementation was indeed unclear since the information was scattered among sections *4 Unified hardware/software design* and *5 Case study*. We have addressed this with the following changes in the revised manuscript: former sections *4.2.3 Wrapping communication* and *4.3 Implementation flow summary* were moved to a new section *5 Deployment of unified components*, becoming sections 5.1 and 5.3 respectively; added a new section *5.2 Implementation platform*.

   The new section 5.2 describes the execution platform that is later used to deploy the case studies. This description provides details of the run-time support, which includes the information requested by the reviewer about interrupt handling for software calls.

5. Reviewer's comment: *A Trait is a very generic template meta programming technique. It remains unclear how this technique solves the problem of HW/SW communication.*

   In this context, the traits only encapsulate the information that the HLS tool or the compiler need to define if the component itself or its proxy is going to be instantiated. In the answer to comment 2 from Reviewer #1, we provide a better description of the trait concept. We have also extended the explanation of the code snippet in the last paragraph of section 5.1.

6. Reviewer's comment: *In the case-study error bars should be added to measured execution times.*

   We have added error bars to Figures 13 and 15b. The figures now show that the execution time of the scheduler varies significantly according to the number of threads in the system, which is an expected result since we have experimented with 8 threads. This explanation was added to the third paragraph of section 6.1.

7. Reviewer's comment: *The main focus of the case-study is on the comparison of component implementations using C++ with the presented unified C++ approach. The evaluation is quite extensive and could be reduced.*

   We have reduced the case study section by moving the paragraphs that described Figure 11 in the original submission to the new section 5.2. Additionally, in order to comply with the page limit, we

chose to reduce the description of the PABX system in the first paragraph of section 6 and removed the PABX system diagram from Figure 8.

8. Reviewer's comment: *The results presented in Table 5 are hard to understand.*

   Table 5 and its description were improved. The values in ()'s were moved to the *System* column, since these values are used to define the "system" overhead (e.g. the memory footprint of the run-time support, FPGA area of the RTSNoC interconnect, etc.), while the *Total* column shows the total area footprint.

Reviewer #3

1. Reviewer's comment: *On the general approachability of the text. The authors seem to assume that readers understand many concepts like "aspect-oriented programming", and do not really explain these concepts in the paper. More explanations on the concepts of OOP/UML and C++ template will be helpful. I understand that many basic concepts take much space to explain. Yet the author should at least say something like "we assume that the readers are comfortable with C++ templates and UML diagrams in the reset of this paper. For information on these, please see [xx] and [yy] for more explanation."*

   Please refer to the answer to comment 1 from Reviewer #2.

2. Reviewer's comment: *While the proposed method could work well for the purpose of describing hardware and software in a unified way, comparison with other methods for the same purpose is missing. For example, it is possible to use C (with extensions/pragmas) to describe both hardware and software (I know people in the industry doing so), and I believe what static metaprogramming offers can also be achieved using C language constructs, like macros and #ifdef blocks. Discussion on the advantage of the proposed approach should be elaborated, with comparison to possible alternatives (not necessarily experiemtal comparison). For example, one of the advantages I can see is that the proposed approach is more systematic and thus can potentially offer better checking at the syntax level; i.e., a domain-specific compiler may easily inform the designer about missing constructs. Yet, a disadvantage could be that a coding style using static-metaprogramming can be difficult to debug. Comparisons like this could help the reader better understand the proposed method and appreciate its unique advantages.*

   Indeed, old CPP macros are still widely used in practice and it is important to provide this kind of comparison in the paper. We would like to thank the reviewer for the suggestion. The second and third paragraphs of section 4.4 address these points.

3. Reviewer's comment: *There are more differences in hardware design and software design that may need attention. For example, for high-level synthesis, bit-accurate data types are very useful. By using only enough bit widths for operators, storage elements and interconnects, significant saving in resource/power can be achieved. On the other hand, software compilers usually only use 8/16/32/64 bit for integers. The tool used in your experiments, Catapult-C, supports the ac_int/ac_fixed datatypes in C++. I wonder if bit-accurate datatypes are used in your experimental evaluation? How do you unify the hardware/software description if bit-accurate datatypes are needed? My personal experience is that there can be subtle issues when overflow occurs.*

   We have implemented our current components using the standard C++ data types (for instance, some of the data types can be seen in the UML diagrams in figures 9, 10, and 11) to ensure maximum flexibility of the unified code. However, we agree that bit-accurate data types are crucial to optimize the area of the final hardware design. We have addressed this point in the fifth paragraph of section 4.2.1. A possible way to provide bit accuracy while keeping a certain degree of uniformity is to use C++ typedef statements to define all allowed data types according to the target domain. For instance, a 5-bit integer can be defined in hardware using `typedef ac_int<5> int5` and in software using `typedef char int5`, since `char` is the smallest native type with the required width. We believe this is a reasonable solution since most compilers and processor architectures support only 8/16/32/64-bit integer types, therefore true bit-accuracy in software would require additional shifting and masking operations that may add a significant overhead.

4. Reviewer's comment: *The approach, as a coding style, can be applied to a more general class of "module selection" problem. The scenarios considered do not have to be limited to either "software" or "hardware". In high-level synthesis, the design space can be explored by using different pragmas (like loop unrolling/pipeling), and each implementation could be a scenario. Some of the techniques described in this paper can actually encode multiple sets of implementation options in a unified description.*

Although we consider that tuning the hardware microarchitecture using synthesis directives is part of the design space exploration and out of the scope of this paper; this is indeed an interesting direction for future works. The hardware scenario could be extended to also provide such kind of semantics in order to aid the design space exploration process. A possible approach would be, for instance, to specify synthesis directives using C's pragmas within an aspect. This aspect would have to be specialized for each component, HLS tool, and intended hardware microarchitecture, since each of these would require a different set of pragmas.

The last paragraph of section 4.4 address the points highlighted above.

Sincerely,

Tiago Rogério Mück and Antônio Augusto Fröhlich

Unified Design of Hardware and Software Components

Tiago Rogério Mück, *Member, IEEE,* and Antônio Augusto Fröhlich, *Member, IEEE,*

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 Sections 4 and 5 we discuss the characteristics that are part of software and hardware scenarios and present our approach for their proper separation and implementation; Section 6 describes our case studies and shows our experimental results; and Section 7 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* (PBD) [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 [18] and the execution of the models for their verification. The authors have elaborated over *SysML*, an extension of UML for system-level design [19], 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 [20] 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 [21]. 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) [22] 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) [23] 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 [24], the ReconOS [25] and the BORPH operating system [26] 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* [27] 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

In principle, the decomposition of a given domain can be obtained following the guidelines of *Object-Oriented Decomposition* [28]. This process results in a model composed by a set of interacting *objects*. Each object is characterized by its *class*. A class defines the structure (the data elements of the object, or its *attributes*) and behavior (through *methods*) for a set of objects. Classes are composed and extended, resulting in a *class hierarchy* that defines the static structure of the system and the binding between objects.

The *Application-driven Embedded System Design* (ADESD) methodology [6] elaborates on this decomposition strategy to add the concept of *aspect* identification and separation at early stages of design. Variability analysis, as carried out in object-oriented decomposition, does not emphasizes the differentiation of variations that belong to the basic behavior of a component from those that emanate from the execution scenarios being considered for it. Components that incorporate environmental dependencies have a smaller chance of being reused in new scenarios, and, given that an application-driven system will be confronted with a new scenario virtually every time a new application is defined, allowing such dependencies could severely hamper the system.

Nevertheless, one can reduce such dependencies by applying the key concept of *aspect-oriented programming* (AOP) [29], i.e. aspect separation, to the decomposition process. Scenario dependencies can be encapsulated in special constructs called *aspects*. An aspect weaving semantic is then defined to describe when and how aspects are applied to components. By doing so, one can tell variations that shape new components from those that yield scenario aspects. For instance, instead of modeling a new component for a family of communication mechanisms that is able to operate in the presence of multiple threads, one could model multithreading as a scenario aspect that, when activated, would lock the communication mechanism (or some of its operations) in a critical section.

The aspect weaving semantics in ADESD is defined by constructs called *Scenario adapters* [30]. 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. These actions may take place respectively before and after each of the component's operation in order to setup the conditions required by the scenario. Figure 1 shows the general structure of a scenario adapter. A *Scenario* construct represents the execution scenario and aggregates all aspect programs that characterize it. The scenario adapter then applies the scenario to the target component. As can be seen in the following sections, scenario adapters can be fully implemented in C++ using its OOP and template metaprogramming capabilities[1]. In contrast to traditional aspect weaving approaches, such as AspectC++ [36], scenario adapters do not require external tool support and can be compiled by any standard C++ compiler.

3.1 ADESD for HW/SW design

In ADESD, knowledge about implementation details should be driven to identify and isolate scenario aspects. In general, aspects such as identification, sharing, authentication, and encryption can be represented as scenario aspects. Such aspects are part of the *application domain*, since they describe behavior related to the application functional requirements. However, the design of new components in a HLS-capable system-level environment must also take into considerations the *platform domain*, therefore yielding at least two additional scenarios: a *software scenario*, in which the component is deployed as part of the software running in a processor; and a *hardware scenario*, in which the component is deployed as dedicated hardware.

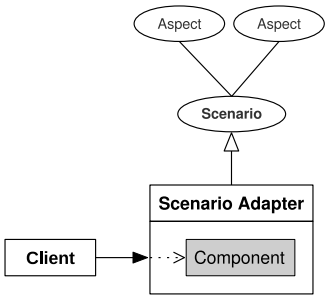

Figure 1. Component adaptation using a scenario adapter.

An initial work [37] has already shown how the ADESD methodology was used to provide a C++ description of a resource scheduler suitable for high-level hardware synthesis. In this paper we further extend the previous case in order to show how ADESD's aspect separation mechanisms can be used to yield components susceptible to both hardware and software synthesis. We demonstrate how characteristics specific of a hardware or a software scenario can be isolated from the core behavior and the interface of a component. By following our design guidelines, the same component description can be tailored by the scenario adapter in order to feed either a software compilation flow or a hardware HLS 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 have 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

1. this paper assumes prior knowledge about OOP, UML and C++. An overview of the relevant concepts is available at [31], [32], and [33]. The reader may also refer to [34], [7], and [35] for an indepth explanation.

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*. Later 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. To illustrate some differences between OOP and TLM-like approaches, Figure 2 shows a simple system modeled using both strategies. The OOP model is more expressive in the way components interactions are can be defined. A component can be either implemented as a global object (*C1* and *C3*) or as a part of another object (*C2*). On the other hand, 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 to the designer.

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[38]. However, this is not the focus of our current work. We aim at providing more general guidelines for describing components.

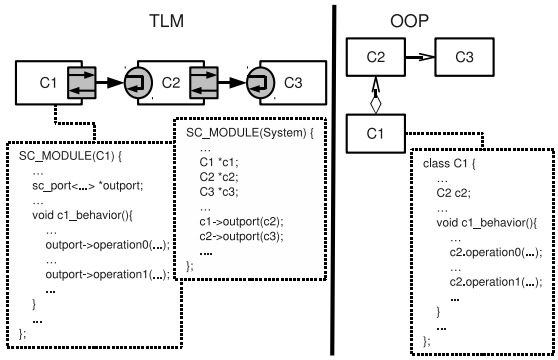

Figure 2. Communication between components in transaction level (left) and object-oriented (right) models.

4.2 Defining C++ unified descriptions

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. These characteristics will be later incorporated to the components using an aspect weaving process. This must be taken into account when implementing the component, otherwise such adaptation is not possible. As mentioned previously, in this paper we consider the following basic differences between hardware and software: *communication interface* and *resource allocation*.

By assuming that a component internal state can be modified solely through local method invocation (e.g. no shared memory between components), the adaptation of the communication interface is straightforward. From the software perspective, the method call interface of a component can be used directly without adaptations. From the hardware perspective, it must be adapted to match the requirements of the chosen HLS tool. Section 4.3 provides more details about the encapsulation of the method call interface.

On the other hand, the encapsulation of resource allocation as an aspect requires a design that allows this concern to be handled externally. To illustrate such design, 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) [39] is a case study on which the design artifacts proposed by ADESD were implemented and validated. The complex behavior of the scheduler motivated its choice as our main case study. Task scheduling involves both synchronous (e.g. creating a new thread) and asynchronous (e.g. preempting the execution of another component) operations. Furthermore, enabling a hardware-implemented scheduler allows a system with less jitter and better support for real-time

applications [40]. Figure 3 shows the entities identified in the domain of real-time scheduling.

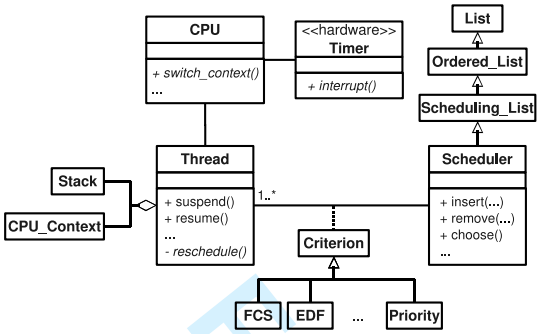

Figure 3. Simplified UML class diagram of scheduling-related classes in EPOS.

In this design, a task is represented by the class *Thread* which defines the execution flow of the task, implementing the traditional functionality of such kind of abstraction (e.g. timing interrupt handling and context switching). Concerns such as timing management are performed through OS abstractions for CPU-dependent hardware (e.g. the *Timer* class).

The classes *Scheduler* and *Criterion* define the structure that realizes the task scheduling itself. Traditional implementations of scheduling algorithms are usually done by a hierarchy of specialized classes of an abstract *Scheduler* class, which can be further specialized to bring new scheduling policies to the system. In order to reduce the complexity of code maintenance (generally present in such hierarchy of specialized classes), as well as to promote its reuse, our design detaches the scheduling policy from its mechanisms (lists implementations) and also detaches the scheduling policy from the thread it represents. The separation of the mechanism from the scheduling policy is what in fact allows the deployment of the scheduler as both hardware and software. The unified description of the *Scheduler* component implements only the mechanisms that realize the ordering of the tasks, based on the selected policy. In this sense, the same component can realize distinct policies, as the definition of the policy is confined in the *Criterion* component, which isolates the comparison algorithm between the elements of the scheduler queue.

The *List* class in Figure 3 is a linked list used to implement the scheduling queues but it is not responsible for the allocation of memory for links and the objects it stores. It implements only the list algorithms and deals with references to such elements. As can be seen in Section 4.3.1, this is what allows us to handle memory allocation as a separated aspect according to the target implementation scenario.

4.2.1 Synthesis considerations

Additional considerations must also be taken into account in order to produce implementations that can serve as input for both hardware and software implementation flows. These are not due to specific hardware or software characteristics, but due to some characteristics of high-level synthesis [37]. One of these characteristics is related to the use of C++ *pointers*. Pointers have no intrinsic meaning in hardware. They are mapped by HLS tools to indices of the storage structures to which they point or to objects that can be statically determined. Thus, *no null or otherwise invalid pointers* are allowed in the source code. However, it is a common coding style to use null pointers to report failures. To avoid this, one can change 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 have 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 4.3 we show how resource allocation can be implemented as a separated *aspect*, however recursion and polymorphism can be implemented using static metaprogramming techniques and C++ templates, 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) [41], and allows the static resolution of calls to virtual methods in the base class.

Another issue that needs attention is the use of bit-accurate data types. HLS tools usually provide specific libraries (e.g. the *ac_int/ac_fixed* from CatapultC [1]) that allow the programmer to use only the necessary bit width for each algorithm. However, while bit-accurate data types yield more efficient hardware, their use in software adds overhead since most compilers and processor architectures support only 8/16/32/64-bit integer types, thus requiring additional shifting and masking operations to emulate bit-accurate behavior. In our case studies we use only the standard C++ data types for

maximum flexibility of the unified code. However, a possible way to provide bit accuracy while keeping a certain degree of uniformity is to use C++ *typedef* statements to define all allowed data types according to the target domain. For instance, a 5-bit integer can be defined in hardware using `typedef ac_int<5> int5` and in software using `typedef char int5`, since `char` is the smallest native type with the required width. Nonetheless, one must take extra care to keep a semantic consistency when mixing types with different widths.

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.

4.3 HW/SW aspects encapsulation

We have previously 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. As mentioned in Section 3, the set of aspects which are part of a scenario are incorporated by a class that defines the scenario itself. The scenario adapter then redefines the operations of the base components, wrapping the original behavior with scenario-specific operations. Each part of Figure 4 is explained in more details below.

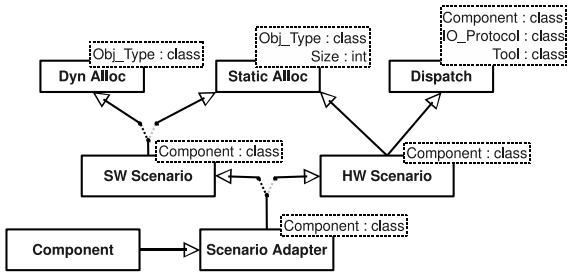

Figure 4. Software and hardware aspect weaving using a scenario adapter

4.3.1 HW/SW 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*. All components operations go through the allocator which reserves and releases storage space on demand. For example, the *insert* and *remove* methods of the *Scheduler* class (Figure 3) are redefined in its scenario adapter as follows:

```
Link insert(Object obj, Criterion crit) {
    Link link = Scenario::allocate(obj, crit);
    Component::insert(Scenario::get(link));
    return link;
}
Object remove(Link link) {
    Object obj = Component::remove(Scenario::get(link));
    Scenario::free(link);
    return obj;
}
```

`Scenario::allocate(obj,crit)` creates an element for the scheduling list and receives as argument the object being scheduled and its scheduling criterion. After the list element is created, it is obtained through `Scenario::get` and inserted in the scheduler (referenced using the `Component` alias). In hardware implementations, `Scenario::*` methods 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. The code sample below summarizes the implementation of the *Static Alloc* aspect. `allocate` methods are also templates to allow the forwarding of different arguments to the allocated object's constructor.

```
template<typename Obj_Type, int size>
class Static_Alloc {
...
    Link allocate() {...}

    template<typename T0>
    Link allocate(T0 &t0) {
        Link link;
        //link is set by the search method
        bool result = search(link);
        if(result){
            //this array store the components
            elements[idx] = Obj_Type(t0);
        }
        return result;
    }

    template<typename T0, typename T1>
    Link allocate(T0 &t0, T1 &t1) {...}
...
    void free(Link link) {
        //just set the corresponding location as available
        alloc_bitmap[idx] = false;
    }

    Obj_Type* get(Link link){
        return &elements[link];
    }
...
};
```

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. A similar approach that uses external allocators for containers such as lists is provided by the *C++ standard template library* (STL) [42]. In principle, we could have relied on STL, however current STL implementations are not synthesizable.

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 (a dispatch mechanism is

not necessary in the software scenario since operations are requested using direct method calls). In Calypto's CatapultC [1] and Xilinx's AutoESL [5], for instance, the top-level interface of the resulting hardware block (port directions and sizes) can be 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. Some tools require the definition of the entry point as a SystemC module with signal ports used to define the IO protocol. Our current implementation is compliant only with CatapultC, since it is the tool used in our experimental evaluation. Nevertheless, the *Dispatch* aspect can be specialized to support different entry-point requirements and different IO protocols (e.g. two-way handshaking, bus-based, etc.). Upon system generation, the desired dispatcher can then be selected.

4.3.2 Scenario definition

The aggregation of aspects in the scenario is performed using a conditional metaprogrammed inheritance. This is shown using the '⋱' notation in Figure 4, which denotes that only one of the two generalizations occur at the same time. The code below shows how the conditional inheritance is defined in the generic implementation of the *SW scenario*:

```
template <typename Component> public SW_Scenario :
    public
        IF<Traits<SW_Scenario<Component> >::static_alloc,
            Static_Alloc,
            Dyn_Alloc>::Result
```

Using the component as a template parameter for the scenario allows one to provide specialization of the scenario for specific components when required. Now considering the generic definition above, the base aspect of the scenario is the result of the `IF` metaprogram which depends on static configurations defined in special template classes called *Traits*. Trait classes provide a convenient way to associate related types, values, and functions with a template parameter type without requiring their definition as members of the type [43]. The code sample below shows the *Trait* class used above, along with the implementation of the `IF` metaprogram:

```
//Default configuration for the software scenario
template <> struct Traits<SW_Scenario<void> > {
    static const bool static_alloc = false;
};
//Specific configuration applied to 'Component'
template <> struct Traits<SW_Scenario<Component> >:
                  public Traits<SW_Scenario<void> > {
    static const bool static_alloc = true;
};

//IF metaprogram
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; };
```

The specialization of *Traits* for *SW_Scenario<void>* defines the default configuration for the software scenario, which is the use of dynamic allocation (`static_alloc = false`). The specialization for *SW_Scenario<Component>* inherits the default configurations but redefines `static_alloc` to `true`, in other words, static allocation will be used instead of dynamic allocation when the software scenario is applied to *Component*. The *IF* metaprogram that choses between possible options is implemented using C++ partial template specialization. If `condition` is `true`, it defines `Result` as the type `Then`. This is encoded in the base definition of the template. The template is partially specialized for the false condition, defining `Result` as the type `Else`.

4.3.3 Scenario adapter definition

The scenario adapter incorporates the hardware/software aspects using conditional inheritance as well. The code sample below shows the generic declaration of the scenario adapter:

```
template <typename Component> class Scenario_Adapter :
    private
        IF<Traits<Component>::hardware,
            HW_Scenario<Component>,
            SW_Scenario<Component> >::Result,
    private
        Component
{
   ...
};

//Trait class for a component
template <> struct Traits<Component> {
    static const bool hardware = true;
};
```

The adapter inherits the component behavior directly from its unified implementation, while the base scenario is chosen by the *IF* metaprogram according to the component's traits. Components that can be implemented as either hardware or software have a trait called `hardware` that is used to define to which domain the component should be adapted. The body of the scenario adapter implementation contains the necessary adaptations of the components methods as described in Section 4.3.1.

4.4 Summary and discussion

We have shown that a single C++ implementation can be used to generate both hardware and software provided that it follows a careful design process. By careful design process, we mean a process that assumes the later weaving of hardware/software dependencies using a scenario adapter. These dependencies and their influence in the unified implementation are summarized below:

Communication interface: limiting the components interactions to method calls allow the creation of a generic external mechanism (the dispatch aspect) that can be used by HLS tools to infer the final hardware interface.

Resource allocation: components that require resource allocation can be designed in order to deal with links to the expected resources, as shown in the implementation of the lists in Section 4.2. The actual allocation can

then be performed externally using the most suitable approach for the target domain.

In this work we have applied the concept of aspects and scenario adapters to introduce hardware/software dependencies and implemented the mechanisms using C++ template metaprogramming. However, other methods can be used for the same purpose. For instance, C's preprocessor macros (e.g. *#define* and *#ifdef* blocks) allow one to introduce software- or hardware-dependent code and to switch between different implementations. This a common approach to distinguish executable from synthesizable C++ in current industrial designs. Nevertheless, C++ templates offer two significant advantages over old-style C macros: 1) macros are basically a text replacement mechanism and do not offer any kind of syntax and type checking, whereas templates are type-safe and its misuse generates compile-time errors; 2) the implementation of a template can be partially or fully specialized to different types. Template specialization is what in fact allows the metaprogrammed mechanism shown in the previous sections. Metaprogramming using C++ templates offers some disadvantages, however. Since templates are parsed internally by the C++ compiler, debugging template metaprograms would require a debugger for the C++ compilation process itself. One must rely only on source code analysis and error messages issued by the compiler.

Another approach is to fully rely on mechanisms implemented in EDA tools to generate both hardware and software from a single input model [15], [20], [22]. Although this usually provides a more automated design flow, it creates a strong dependence between the source code and very specific tools. Providing a systematic way to describe the semantics of the design process in the source code level reduces such limitations. ANSI C++ and its OOP features are extensively supported by hardware HLS tools and software compilers, thus increasing the applicability of our approach over different design flows and tools.

It is also worth recalling that, as mentioned in Section 4.2.1, in HLS, the design space can also be explored by using different synthesis directives (e.g. loop unrolling/pipelining), yielding different implementations for the same C++ construct. Although we do not deal with synthesis directives in this work, the techniques described in this paper can be used to encode multiple sets of implementation options. For instance, directives defined using *pragmas*[2] can be specified within an aspect. Indeed, this aspect would have to be specialized for each component, HLS tool, and intended hardware microarchitecture, since each of these would require a different set of pragmas.

2. the *#pragma* directive is the method specified by the C standard for providing specific additional information to the compiler. It is platform and compiler dependent.

5 Deployment of Unified Components

A uniform communication infrastructure is a requirement for design strategies in which the same component can have different implementations in different domains. In this section we describe our approach to support the seamless communication between hardware and software and the current design flow used to deploy our unified components.

5.1 Wrapping communication

As discussed in Section 4.1, the component *C2*, shown in the OO model in Figure 2, is an attribute of *C1*. Considering that *C1* is the top-level in a HLS/compilation step, *C2* would be compiled/synthesized together with *C1*. However, if the designer intends to have *C1* and *C2* as two independent components implemented in different domains, then an additional mechanism is necessary.

A way to overcome this issue is to use concepts from distributed object platforms [27]. Figure 5 illustrates possible cross-domain 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 explicitly defined in the *SW->HW* interaction since the *dispatcher aspect* can already play its role for components in hardware.

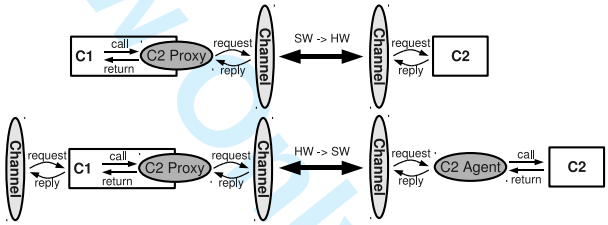

Figure 5. Communication between components in different domains. Leftmost components request operations from the rightmost ones (scenario adapters are omitted for simplicity).

A key 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:

```
//C2 definition in software-ready C++
class C2 :
  public IF<Traits<C2>::hardware,
            HW_Proxy<C2>, Scenario_Adapter<C2> >::Result {};

//C2 definition in hardware-ready C++
class C2 :
  public IF<Traits<C2>::hardware,
            Scenario_Adapter<C2>, SW_Proxy<C2> >::Result {};
```

In the final implementation domain, *C2* can be defined as an empty class that inherits from its actual implementation depending on the configuration defined by *C2*'s traits. In the example above, *class C2* instances in the software domain are instances of *Scenario_Adapter<C2>* when *Traits<C2>::hardware* is *false* (*Scenario_Adapter<C2>* is also adapting C2's implementation to the software scenario). Otherwise, all instances map to *HW_Proxy<C2>*, which implements the proxy for the actual implementation of *C2* in the hardware domain.

5.2 Implementation platform

There are 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)). Figure 6 shows the general structure of the chosen hardware/software architecture to implement these mechanisms. We rely on a NoC as the main communication link between hardware and software components. The *Real-time Star Network-on-Chip* (RTSNoC) [44] is the core component of our SoC platform. RTSNoC consists in a set of *routers* with a star topology that can be arranged forming a 2-D mesh. Each router has eight bidirectional channels that can be connected to cores or to channels of other routers. Each hardware component is deployed as a node connected to a router. Software components are compiled with an RTOS and run in a *CPU node* that consists of a softcore CPU and memories. Shared IO peripherals are encapsulated in separated *IO nodes* so they can be used by both hardware and software components.

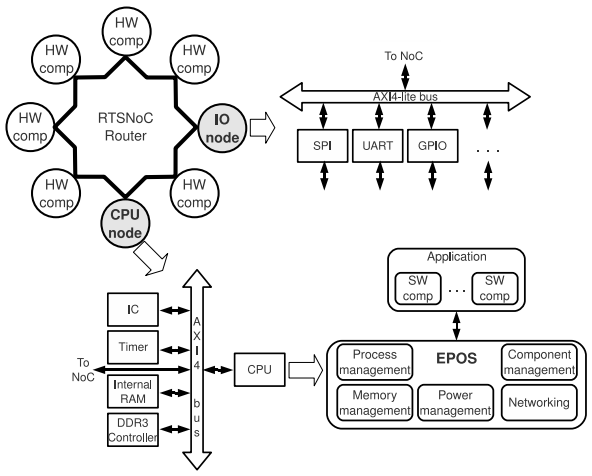

Figure 6. System-on-Chip platform

The internal structure of CPU and IO nodes is based on the AXI4 [45] family of protocols, which is becoming the industry's standard for bus-based interconnection. In our current implementation we have relied on IPs available at Opencores [46]. Our current CPU node, for instance, is based on the *Plasma* softcore, an implementation of the MIPS32 ISA.

The software components run on EPOS [39]. EPOS provides the necessary run-time support to implement the proxies and agents described previously. Figure 6 summarizes the main families of abstractions provided by EPOS to applications. In a SW->HW interaction, calls to hardware components are converted by the proxies to remote call requests handled by the component manager (*Component Management* abstraction). The component manager was added to EPOS as the main software infrastructure to handle the communication between software and hardware components. It keeps lists of all existing proxies to hardware and agents to software. Each component is associated to a unique ID that is mapped by a static resource table to a physical address in the NoC. Upon a call request, this address is used to build packets containing the target method ID and its arguments. These packets are sent through the NoC using the OS networking stack. If the method has return values, the component manager blocks until it receives the return values from the hardware component.

On the hardware side, the entry-point defined by the *dispatch aspect* (Section 4.3) blocks until a packet containing a method ID is received, followed by packets containing the arguments. It parses the packets and performs the local method invocation, sending back through the NoC eventual return values.

Packets sent to the CPU node trigger interrupts that are handled by an ISR defined by the component manager. The ISR reads all pending packets and performs the necessary operations. When the manager receives a packet containing return values, it searches a list of blocked proxies and forwards the packet to the correct one. When a packet contains data from a method call request, the information is forwarded to the respective agent. The method dispatching of a software agent is very similar to the entry-point definition in the hardware dispatch aspect, but it receives data through successive explicit calls. Once all the data is received, the local call is performed in the software component.

5.3 Design flow summary

This work focuses only on general implementation guidelines and not on a whole design flow. Issues such as design space exploration and design verification are not in the scope of this work. Nevertheless, our mechanisms based on standard C++ solutions allow for a straightforward integration with current synthesis tools and compilers. Figure 7 shows the steps we are currently using to go from C++ unified descriptions to components deployed in a physical platform.

The first steps show the artifacts of our proposed approach. Aspects, scenarios, and the unified implementation of components made up the inputs for the implementation flows. Proxies and agents can be automatically generated from the components' interface by

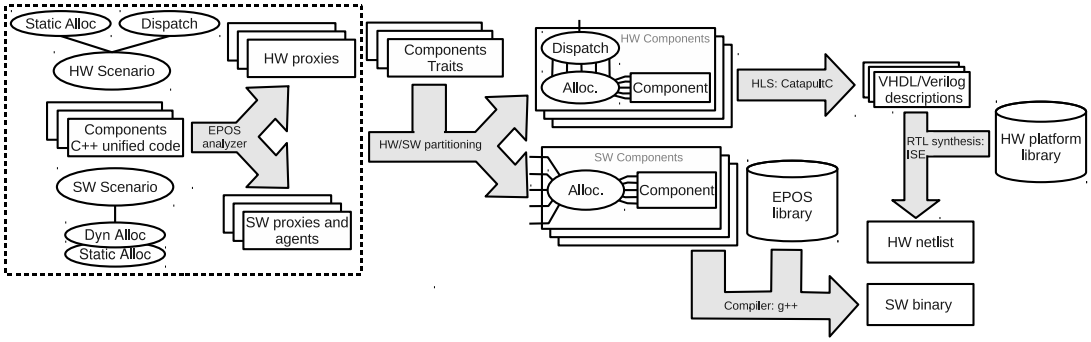

Figure 7. Summary of the implementation flow

a tool (EPOS's syntactical analyzer [47] 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 run-time support implemented in EPOS using the GCC C++ compiler. In the hardware flow, CatapultC 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 hardware IPs described in Section 5.2.

6 Case Study

In order to evaluate our approach, we have analyzed an industrial PABX application and designed some of its basic building blocks using the proposed implementation guidelines. The main component of the PABX system 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 8 shows the block diagram of the digital part of the PABX system, with is deployed as a SoC in an FPGA. The components we have selected to be reimplemented using unified C++ are highlighted and appear in as hardware and software, since they can move between both domains depending on the final hardware/software partitioning. These components are described in more detail below:

EPOS's scheduler: the case described in Section 4.2. Figure 9 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 [48] that is used to reduce the traffic of data transmitted through the network. It performs data compression using an *adaptive differential pulse-code modulation* (ADPCM) algorithm to convert 16-bit samples to 4-bit samples. Figure 10 shows the structure of the codec. The encoding and decoding algorithms

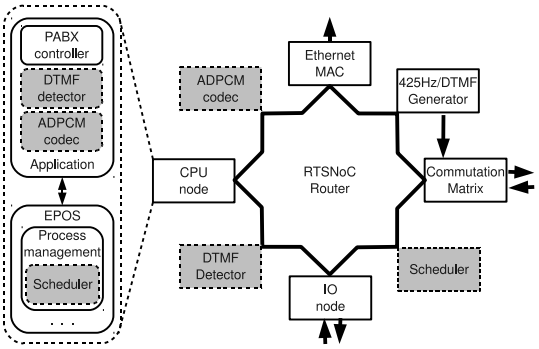

Figure 8. PABX SoC block diagram

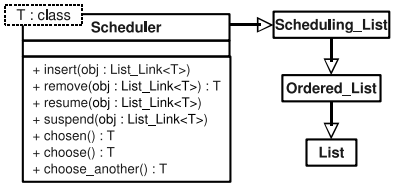

Figure 9. Scheduler component definition

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.

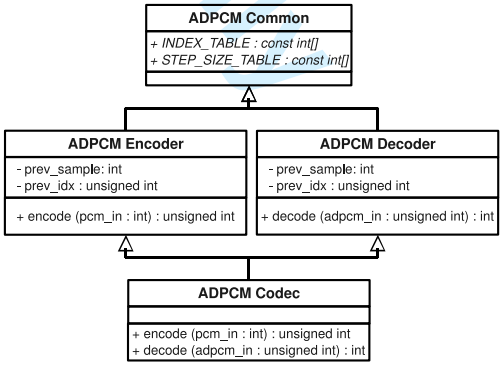

Figure 10. ADPCM codec component definition

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 11 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 pseudo-code in Figure 11 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.

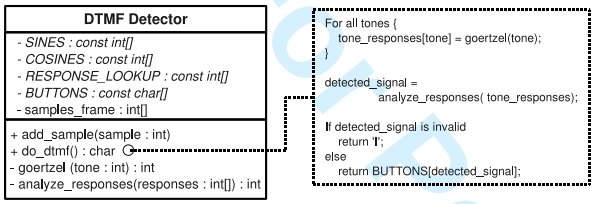

Figure 11. DTMF detector component definition

6.1 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 time-stamp 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.

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. The execution time of the scheduler varies significantly according to the number of threads in the system (see error bars in Figure 13). In this evaluation, we have experimented with 8 threads and a round-robin scheduling criterion. For the DTMF detector, the

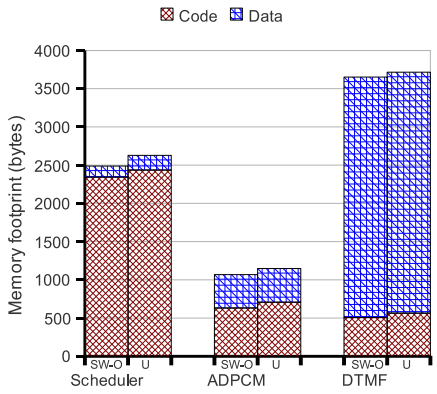

Figure 12. Memory footprint of software-only C++ (SW-O) *vs.* Unified C++ adapted to the software domain (U)

Table 2
Execution time of software-only C++ *vs.* unified C++

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

difference increases to 5%. The original DTMF detector requires a single call to *do_dtmf* to analyze a frame of samples, 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 UV 2011a* was used to obtain RTL descriptions of the components. The descriptions were then synthesized using *Xilinx's ISE 13.4* targeting a *Virtex6*

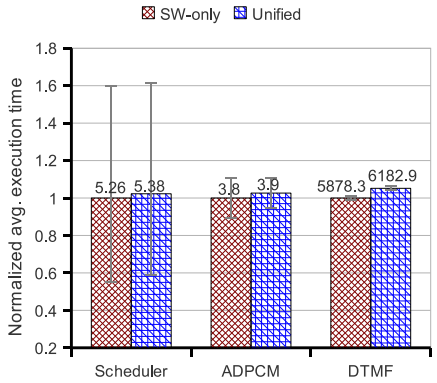

Figure 13. Normalized average execution times. Absolute values are shown above their respective bars.

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 C++ *vs.* unified C++

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

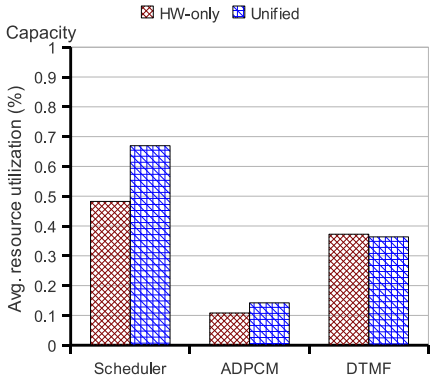

Figure 14. Average FPGA resource utilization

Table 3 shows the amount of FPGA resources required for each component and Figure 14 plots the *average 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 ADPCM 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 5.1) 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 the communication infrastructure required to handle transparent hardware/software communication. EPOS's internal components are implemented using static metaprogramming to achieve high reusability with 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*—Section 5.2, which implements the runtime support). 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 in the *System* column are the total footprints subtracted by the footprints of all case studies in the respective domain. For instance, the software footprint shown in the *System* column for 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 values in the *System* column, 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 proxy, we estimate an initial overhead of about 4 Kbytes plus 30 bytes for each additional operation. We expect to significantly reduce this overhead in future implementations, however, the current results are acceptable if compared with similar infrastructures [49]. 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%).

7 Conclusion

In this paper we have explored a methodology based on AOP and OOP concepts in order to produce unified

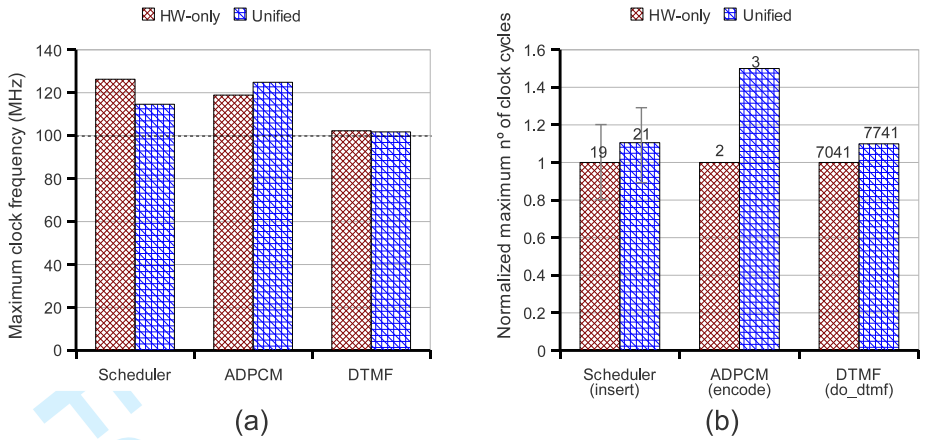

Figure 15. Performance of hardware-only C++ *vs.* unified C++. In Figure (b), values are normalized to compare all components in the same scale.

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

Table 5
SoC area footprint. In the hybrid partitioning, only the Scheduler is in software.

Partitioning	Memory (code+data)		FPGA Avg. rsc. util.	
	Total	System	Total	System
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%

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 a 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 Pizani Flor and Marcelo Berejuck for providing the original implementation of the some of the case studies. This work is also partially supported by 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 Int. Conf. on Emerging Technologies and Factory Automation*, Catania, Italy, 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 Conf. and Exhibition*, vol. 2, 2004, pp. 1132–1137.

[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, 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 Conf. & 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, 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 Conf. 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 Conf. 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 Conf. on Design, Automation and Test in Europe*, Dresden, Germany, 2010, pp. 1201–1206.
[18] OMG, *Unified Modeling Language (UML)*, 2008. [Online]. Available: http://www.uml.org
[19] ——, *OMG Systems Modeling Language (SysML)*, 2010. [Online]. Available: http://www.sysml.org
[20] 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, 2009.
[21] 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.
[22] 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*, 2008.
[23] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proc. of the 1st IEEE/ACM/IFIP Int. Conf. on Hardware/software codesign and system synthesis*, Newport Beach, USA, 2003, pp. 19–24.
[24] 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 Int. Conf. on Field Programmable Logic and Applications.*, 2007, pp. 98 –103.
[25] E. Lubbers and M. Platzner, "A portable abstraction layer for hardware threads," in *Field Programmable Logic and Applications, 2008. FPL 2008. Int. Conf. on*, 2008, pp. 17 –22.
[26] 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, 2008.
[27] 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.
[28] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston, *Object-oriented analysis and design with applications*. Addison-Wesley, 2007.
[29] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proc. of the European Conf. on Object-oriented Programming*, vol. 1241, Jyväskylä, Finland, 1997, pp. 220–242.
[30] 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.
[31] "Wikipedia — Class diagram," 2012, http://en.wikipedia.org/wiki/Class_diagram.
[32] "Wikipedia — Object-oriented programming," 2012, http://en.wikipedia.org/wiki/Object-oriented_programming.
[33] The C++ Resources Network, "Templates," 2012, http://www.cplusplus.com/doc/tutorial/templates/.
[34] C. Larman, *Applying UML And Patterns: An Introduction To Object-Oriented Analysis And Design And Iterative Development*. Prentice Hall PTR, 2005.
[35] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, ser. C++ in-Depth Series. Addison-Wesley, 2001.
[36] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: an aspect-oriented extension to the C++ programming language," in *Proc. of the Fortieth Int. Conf. on Tools Pacific: Objects for internet, mobile and embedded applications*, Sydney, Australia, 2002, pp. 53–60.
[37] 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 Int. Conf. on Electronics, Circuits, and Systems*, Beirut, Lebanon, 2011, pp. 736–739.
[38] N. Abel, "Design and Implementation of an Object-Oriented Framework for Dynamic Partial Reconfiguration," in *Proc. of the Int. Conf. on Field Programmable Logic and Applications*, 2010, pp. 240 –243.
[39] The EPOS Project, "Embedded Parallel Operating System," 2011, http://epos.lisha.ufsc.br/.
[40] 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 Int. Conf. on Computational Science and Engineering - Volume 02*, Washington, USA, 2009, pp. 382–387.
[41] J. O. Coplien, "Curiously recurring template patterns," *C++ Rep.*, vol. 7, pp. 24–27, 1995.
[42] A. Stepanov and M. Lee, "The Standard Template Library," HP Laboratories, Tech. Rep., 1995.
[43] N. C. Myers, "Traits: a new and useful template technique," *C++ Report*, June 1995. [Online]. Available: http://www.cantrip.org/traits.html
[44] 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.
[45] IEEE, "AMBA AXI Protocol Specification (Rev 2.0)," 2010, http://www.arm.com.
[46] "Opencores," April 2011, http://opencores.org/.
[47] 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.
[48] 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
[49] G. Gracioli and A. A. Fröhlich, "ELUS: A dynamic software reconfiguration infrastructure for embedded systems," in *Proc. of the IEEE 17th Int. Conf. on Telecommunications*, april 2010, pp. 981 –988.

Tiago Rogério Mück received his B.Sc. in Computer Science from the Federal University of Santa Catarina, Brazil. He is currently a final year M.Sc. student in the Computer Science Department 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.