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