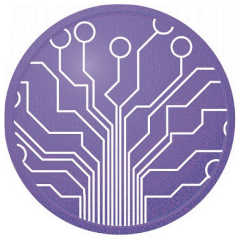


ESLsyn

Proceedings of the 2011 Electronic System Level Synthesis Conference

San Diego, CA, USA
June 5-6, 2011

General Chair:

Dan Gajski
University of California, Irvine

Co-Chair:

Adam Morawiec
ECSI, France

ESLsyn is an ecsi event!

Table of Contents

Table of Contents

ESLsyn 2011

A Hardware/Software Codesign Template Library for Design Space Exploration

Peter Brunmayr, Jan Haase, Christoph Grimm
Institute of Computer Technology,
Vienna University of Technology,
1040 Vienna, Austria
{brunmayr, haase, grimm}@ict.tuwien.ac.at

***Abstract*—The ability to map a high level algorithm either to hardware or software simplifies design space exploration of cyber-physical systems. Thereby, low level tools can be utilized for accurate design parameter estimation, which helps to evaluate the effect of system level design decisions.**

Especially complex data structures pose a problem in this context. The different structure of memory in hardware and software requires different data structure implementations. With the presented data structure library a consistent design flow from a high level system model to either a hardware or software implementation is enabled. The concept extends the idea of abstract data types across the hardware/software boundary. Container adapters with appertaining implementations for system level simulation, hardware and software implementation support the designer throughout the whole design process.

The benefit of the presented library is demonstrated and evaluated by a case study. With very little effort seven different hardware solutions were generated and compared concerning their power consumption and their resource usage.

I. Introduction

The design of cyber-physical systems has to follow different design constraints. To find the best solution early design decisions like the hardware/software partitioning can not be based solely on the designer's experience. To overcome this problem, the evaluation and classification of different solutions, called design space exploration, has emerged as an important system level design technique.

Design space exploration demands not only modeling and simulation techniques at the system level, but also a link to an actual implementation. To accurately classify a solution, low level tools are required to estimate design parameters like power, performance and cost. Furthermore, the translation to a low level implementation is required for a rapid prototype generation.

In [1] the Tripartite system level design approach has been presented. By separating not only computation and communication, but also complex data structures and by the utilization of modern high level synthesis (HLS) tools, directly synthesizable and compilable computation components can be designed in SystemC [2]. This simplifies the way from a realization independent system level description to an actual hardware/software implementation.

The separation of complex data structures results from input restrictions of modern HLS tools. Many abstract data types are based on dynamic memory management, which can not be synthesized by current HLS tools [3]. Another constraint regards the limited support of pointers, which are also frequently used in abstract data types. Furthermore, the different structure of memory in hardware and software requires specific code constructs to efficiently map data structures to memory.

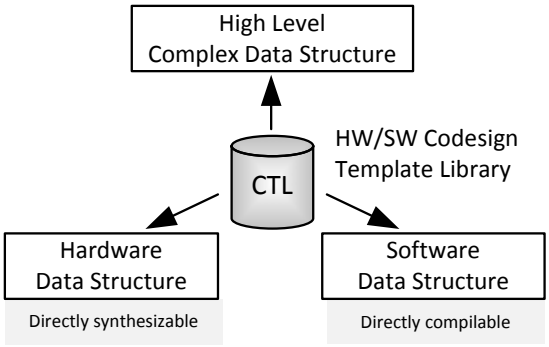


Fig. 1. The codesign template library (CTL).

In this paper the hardware/software codesign template library (CTL) is presented. It enables the usage of high level complex data structures for hardware/software design space exploration. Without rewriting the algorithm, the design can be made synthesizable or compilable by replacing the high level data structure's implementation with specific library components for either hardware (HW) or software (SW), see Fig. 1. For hardware designs, the library additionally provides the opportunity to map data structures to specific memory structures. Furthermore, this introduces the possibility to efficiently use and share the available memory structures by mapping different data structures to different address ranges of a memory structure.

The remainder of this paper is structure as follows: After the presentation of related work in Section II the concept of the Tripartite system level design approach is presented in Section III. The main contribution of this paper, the Hardware/Software Codesign Template Library can be found in Section IV. The CTL and the Tripartite system level design approach are applied to a case study. Different design solutions are compared concerning their power consumption and concerning their resource usage in Section V. Finally, Section VI concludes this work and gives a short outlook.

II. Related Work

In recent years many design space exploration environments have been published. Two examples are Metropolis [4] and PeaCE [5], which are based on the famous codesign pioneer projects Polis and Ptolemy. Both solutions start with a proprietary input format, which reduces industry acceptance, since SystemC emerged as the de facto standard for system level design.

The Daedalus framework presented in [6] is a design framework for system-level architecture exploration, system-level synthesis and prototype generation using a set of tools. It guides the designer from a sequential C program to an heterogeneous multi processor system-on-chip solution. A disadvantage of the Daedalus framework is the restriction to C. Although the framework provides a fast way from an algorithm in pure C to a HW/SW prototype, its modeling capabilities for heterogeneous systems are very limited. Heterogeneous systems which consist of a data flow dominated and a control flow dominated part can not be modeled with the Daedalus framework.

The System-On-Chip Environment is a system level design tool developed at the University of California, Irvine [7]. It supports embedded systems design starting with a specification modeled in SpecC [8]. Interactively guided by the user via a graphical user interface, the specification is refined step by step to an actual HW/SW implementation. The tool allows the exploration of different design solutions at various levels of abstraction and permits the automatic model refinement to lower abstraction levels. The use of SpecC, which is based on ANSI C reduces the design and modeling capabilities. Algorithms, which are modeled using a high level language may operate on complex data structures. To use the design flow in this case, a manual translation to SpecC using only simple data structures is required.

Another refinement based design environment is presented in [9]. This design framework based on SystemC enables hardware/software cosimulation including different levels of abstraction. It also simplifies the generation of different HW/SW prototypes using a library based approach. Tough, it mainly focuses on communication refinement. For the generation of a hardware implementation out of a high level computation component no solution is provided.

Like our approach, Systemcodesigner [10] is based on SystemC and the high level synthesis tool Cynthesizer [3] from ForteDS. The design environment provides a fully automated design space exploration approach. Like all other presented solutions, it does not support complex data structures, which significantly decreases the required abstraction level.

In [11] a synthesizable container library called generic class library has been presented. However unlike the CTL, it does not provide a methodology to exchange the container implementation with specific high level, HW or SW implementations.

III. The Tripartite Design Flow

As its name implies, the core of the Tripartite system level design flow is a threefold separation [1]. If a module is designed at the system level, its components are separated into the three categories: communication, computation and complex data structures. This separation simplifies the translation from the system level model to an actual hardware description language (HDL) or to a C++ implementation, cf. Fig. 2 system level model and HDL/C++ model. Each module can be mapped to either hardware or software.

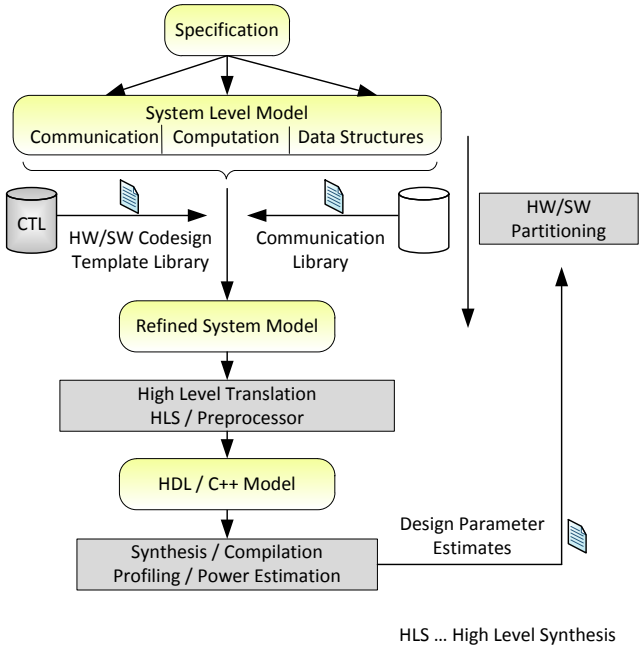


Fig. 2. The Tripartite system level design approach.

By simplifying the step from a system level model to an actual implementation, different solutions for a design decision like the HW/SW partitioning can be evaluated and compared. The HDL/C++ model can be further processed by traditional HW and SW design tools. These low level tools provide more accurate estimates of design parameters like cost, performance or power. The estimates help to classify solution and thereby simplify the process of finding the best solution for a given design decision. By further mapping the HDL/C++ implementation to an FPGA and some processor, a prototype can be generated rapidly.

Typically, system level descriptions are not directly synthesizable and compilable at the same time. Thus, each module has to be refined for either hardware or software. By using the Tripartite system level design approach, an error-prone process of manual code translation is avoided, because it enables the realization independent design of computation components. Communication components and complex data structures are replaced by refined components from a library. This step is shown in Fig. 2 as step from the system level model to the refined system model. The refined software modules can be further translated to pure software by simple

preprocessor directives. The directives are used to transform the SystemC computation module to a pure C++ class. Using a high level synthesis tool, the refined hardware modules can be synthesized. In Fig. 2 the high level synthesis for hardware and the preprocessor directives for software are combined to the "High Level Translation" block. The reasons for the threefold separation are presented in the following:

- **Computation:** The pure computation is very similar implemented in HDLs compared to software programming languages. Under certain conditions, high level synthesis tools support the direct synthesis of untimed computation code. This enables an almost equal description of computation for HW and SW.
- **Communication and Synchronization:** Contrary, an accurate description of communication and synchronization behavior needs a lower abstraction level. To define specific timing relations of e.g. interface protocols, a model at the register transfer level (RTL) is necessary. In software, communication is often implemented using facilities provided by the operating system. Hence, operating system (OS) specific code is necessary to realize communication. These differences illustrate the advantage of the separation of communication and computation.
- **Complex Data Structures:** The separation of complex data structures results from different memory models used for hardware and software design. A software programmer usually assumes a linear infinite memory. Based on this memory model, many complex data types use dynamic memory management and pointer arithmetic, which currently can not be synthesized by HLS tools. A hardware designer has to particularly decide between different memory structures. A design realized on FPGA for example can use external memory, different types of block RAMs or distributed memory. Many HLS tools require special code constructs to map data structures to specific memory structures.

Using the threefold separation into communication, computation and complex data structures the simultaneously synthesizable and compilable design of computation components is enabled. At the same time, the separation accounts for the differences between hardware and software concerning communication and data structures. With a data structure library, realization independent data structures can be used at the system level. During refinement, these data structures are replaced by specific HW or SW components. An example for such a library is the Hardware/Software Codesign Template Library, which is presented in this paper.

IV. The Hardware/Software Codesign Template Library

The Hardware/Software Codesign Template Library is a library consisting of data type independent containers implemented in SystemC. The library concept follows the idea of abstract data types [12] like it is used in the C++ Standard Template Library (STL). Each container has a well-defined interface, which provides operations to add, remove or manipulate data inside the container. Further, properties are specified, which define the complexity of certain operations. The properties help the designer to estimate the algorithm's performance, which depends on the used operations. The actual implementation of the container is not fixed. This idea of abstract data types can be extended across the hardware/software boundary to a system level design concept. The system level designer operates with abstract realization independent containers with certain properties. After HW/SW partitioning, the abstract data type is mapped to an actual hardware or software implementation fulfilling these properties.

The library currently consists of seven elements, which are shown in Tab. I. The simple containers Array and Const Array are not real complex data structures. By adding them to the library, the memory mapping facility, which is presented in more detail later in this Section, can also be utilized for such simple data structures. For all containers an example property is shown in Tab. I. The List for example has to be implemented so that adding and removing elements from anywhere within the List is possible in constant time. A typical implementation for the List is a double linked list. Although the containers are not directly comparable to the Standard Template Library, most of them are influenced by the STL's sequence containers, like Vector, Deque or List. However, the library can easily be extended also to other types of containers.

TABLE I
Container Types of the CTL.

Container	Properties
Array	Accessing individual elements by their position index in constant time.
Const Array	Reading individual elements by their position index in constant time.
Queue	Add elements at the beginning and remove elements from the end in constant time.
Stack	Add and remove elements in constant time at the end.
Vector	Add and remove elements in constant time at the end.
Deque	Add and remove elements in constant time at the beginning and at the end.
List	Add and remove elements in constant time anywhere.

In the following, this Section is divided into two subsections. First the basic structure of the CTL is presented in more detail. After that, the separation mechanism to connect the container implementation to the interface is shown.

A. Basic Structure

As can be seen in Fig. 3, the CTL consists of several sublibraries. The containers used by the system level designer are so called container adapters, which simply provides the defined interface for a specific container. Except for a few functions, which are implemented directly in the container adapter, most function calls are rerouted to the connected container implementation. Hence, a container adapter cannot be used without the container implementation. In general it

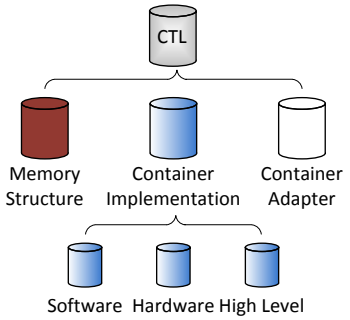


Fig. 3. Basic structure of CTL.

is possible to have several different implementations, which have different advantages and disadvantages. Currently the CTL provides three different container implementations for each container type:

- **High Level:** The high level containers are intended for the use during system level simulation. These containers use standard C++ containers to store the data elements. Additionally, the maximum memory load is logged during simulation. This helps the designer to estimate the required memory for each data structure by simulating worst case application scenarios.
- **Hardware:** The hardware containers only use static memory management and the utilization of pointers is reduced to a minimum. These containers are directly synthesizable. Additionally, a memory mapping facility is provided. Each hardware container has to be connected to a memory structure. In the hardware container, the different interface functions are implemented and reduced to a sequence of simple read and write operations performed on the connected memory structures.
- **Software:** Finally, the software sub-library is a pure C++ library. It is directly compilable by a standard C++ compiler and it internally uses the standard STL components. Certainly, different container implementations for software can be added in the future as well. Due to the overhead of dynamic memory management a static implementation would be conceivable. Especially for the design of systems with tight constraints, such an additional implementation alternative would be a benefit.

The final sub-library is the Memory Structure sub-library. It shall provide typical memory structures available in the target technology. During design refinement, hardware containers are mapped to actual memory structures. Whereby, different data structures can be mapped to the same memory structure. Currently the library focuses mainly on FPGA designs. It provides elements to use the three basic types of internal memory typically available in todays FPGAs: distributed RAM, block RAM and dual-port block RAM. Generally, only containers of the same thread can be mapped to the same data structure. In this case the HLS tool will schedule the memory accesses. If a dual-port block RAM is used, containers of at most two different threads can be mapped to it. Additional logic for scheduling memory accesses is required, if containers of more than two threads are mapped to a dual-port block RAM or if containers of more than one thread are mapped to a single-port block RAM.

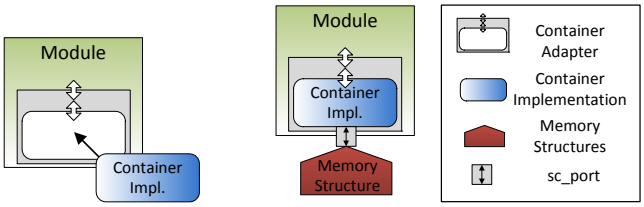
B. Separation Methodology

The principle of the CTL is based on the separation of design components. On the one hand side the container implementation has to be separated from the container adapter to enable the use of different implementations for simulation, hardware and software. On the other hand specifically for hardware, the container implementation should again be separated from the actual memory structure, so that it is possible to map different containers to one data structure, which would increase the efficiency of the resource usage.

Different ways exist to separate design components like the interface and the actual implementation. The basic concept behind it is the polymorphism. With polymorphism the separation of the external shape from its internal form is meant. In C++ two types of polymorphism are available [13]:

- **Dynamic Polymorphism:** The dynamic polymorphism is a key component of object oriented programming. The distinction between static and dynamic is based on the time, when the connection between interface and implementation is resolved. The port and channel concept of SystemC basically uses dynamic polymorphism. The realization is based on the usage of pointers, which are resolved, when a polymorphic function is actually called during runtime. This pointer usage and the dynamic nature is probably the reason why e.g. the ForteDS Cynthesizer only supports one layer of port and channel connection, which limits the usability of dynamic polymorphism for the CTL.
- **Static Polymorphism:** Static polymorphism is based on generic programming, hence on template classes. In this case, the functions operate on some variable, which's type is set via a template parameter. When the class is instantiated, the actual type, hence the actual implementation of this variable is set. In static polymorphism, the interface is defined implicitly by the function calls which are performed on the data type. The connection between implementation and interface is resolved already by the compiler and the connection has to be fixed when the components are instantiated. The fact that the resolution is already preformed by the compiler might also be the reason, why static polymorphism can be easier handled by current HLS tools.

The connection of the container adapter with the container implementation is realized using static polymorphism. Fig. 4(a) shows the usage of the CTL in a computation component. In the module, the computations are performed on the container adapter. The actual implementation is set via a template parameter of the container adapter. If the template parameter of the container adapter is set by a template parameter of the computation module, it is possible to set the actual implementation, when the computation module is instantiated.



(a) Container adapter with implementation

(b) HW implementation connected to a memory structure

Fig. 4. Functionality of CTL.

In this way it is possible to design the computation module without determining the actual implementation of the data structure.

Another point of the CTL where polymorphism is applied is the connection of a hardware specific implementation with a memory structure. Each hardware specific container implementation reduces complex interface calls to simple read and write operations, which are performed on a memory structure connected via an sc_port, see Fig. 4(b).

Although static polymorphism is better supported by current HLS tools, it can not be used in this case. Using static polymorphism would not allow to connect different hardware containers to the same memory structure. Instead, the memory structure would be instantiated separately in each hardware container. Thereby the possibility to map different hardware containers to the same memory structure would get lost. Currently, containers can be mapped to the same data structure if they have the same bit width. Memory conflicts are avoided by an address offset, which maps each container to a different address range. The offset is set at the same time, when the port is mapped to the memory structure.

V. Case Study

To show the applicability of the Tripartite design flow and especially of the CTL, a case study is presented in the following. The basic structure of the system is shown in Fig. 5. The system has been modeled at the system level using the Tripartite design flow. The whole communication and synchronization, both at the inputs and outputs of the submodules and at between them, is realized in communication or input/output channels. Complex data structures are realized using the CTL containers. The computation modules are purely untimed and they do not include any implementation details of communication and data structures.

The presented system consists of three modules: Module A, Module B, and Module C. The first module, Module A, has two inputs and one output. Internally it uses a List to store data. The second module, Module C, is basically a digital filter. The coefficients are stored in a Const Array and the data values are pushed to a Deque. Module C reads the outputs of Module A and B. For the computation, two containers, a Vector and a Queue, are required. The result is written to the only output of the system.

The simulation model for the system level uses high level communication channels. The container implementation of the

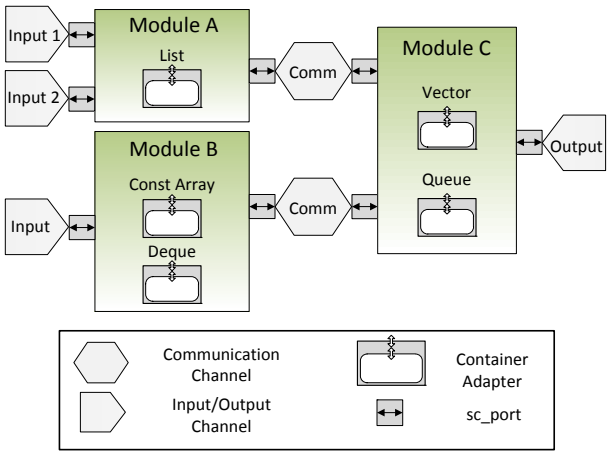


Fig. 5. Basic structure of design example.

used container adapters is set via a template parameter when the computation module is instantiated. Using the high level container implementations, the general functionality of the system is verified and the required memory size is estimated.

By replacing the communication channels with refined software channels and by replacing the high level container implementations, with software implementations, the model can be refined to a software model. Remaining SystemC macros and keywords are replaced using preprocessor directives. The resulting model is a pure C++ model and it can be compiled with any C++ compiler.

The same high level model can be translated to a hardware model, by replacing data structures and communication channels with hardware specific components. The hardware communication channels include cycle accurate I/O protocols. The hardware implementations of the containers use static memory management and the usage of pointers is avoided.

Using the memory mapping capability, each data structure can be mapped to either distributed RAM, block RAM or to a specific port of a dual port block RAM. During design space exploration seven different memory structure mappings have been generated and tested. Each resulting hardware model has been synthesized by the ForteDS Cynthesizer. As target technology, the Xilinx XC4VFX20 Virtex4 FPGA [14] running with 100 MHz has been chosen. The HLS tool generates Verilog code, which has been further synthesized by using the logic synthesis tool Synplify from Synopsis [15]. The resulting netlists have been analyzed using XPower, a power analyzer provided by Xilinx. Results of the power analysis and of the synthesis process are shown in Fig. 6.

The power information is of course only a preliminary estimation. More accurate estimates can only be generated if an exact application scenario is simulated. Thereby information like the toggle rate of single bits can be acquired. However, the generated information provides a first estimate of the power consumption of the different design solutions. Fig. 6 illustrates the relation of the power consumption and the required area in terms of look-up-tables (LUTs). All solutions have been

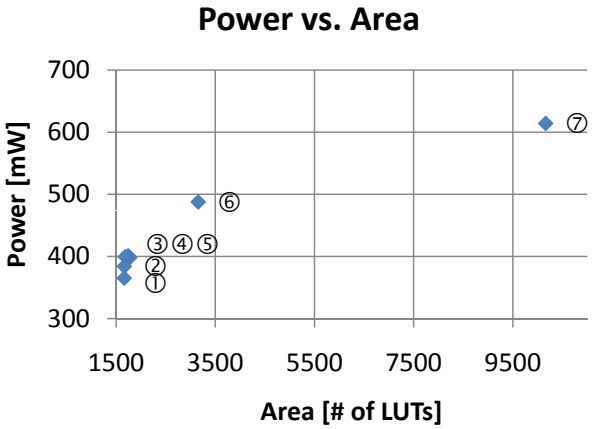


Fig. 6. Comparing power and area of different design solutions.

optimized for area, by using design constraints during HLS also the optimization of e.g. the overall latency is possible.

The design solution with the highest power consumption is solution ⑦. It consumes more than 700 mW and consists of about 10,000 LUTs. In this case all five data structures are mapped to distributed RAM. Obviously, this solution has the shortest latency, since all the data can be accessed in parallel, in contrast to a solution with block RAM, where only one RAM cell can be accessed per cycle. In solution ⑥, the List of Module A is mapped to a block RAM. In this way the power consumption can be reduced by more than 100 mW and the LUTs can be reduced by about 7,000. This is explained by the fact, that the List is with 50 words with 30 bits each one of the largest data structures in the design.

The smallest solution is solution ①. All five data structures are mapped to five separate block RAMs. The design is also the one with the least power consumption. So in general this shows, that mapping data structures to block RAMs will reduce the power consumption. However, there is only a limited number of block RAMs on an FPGA and most block RAMs are only filled partially in this case study. So, it might be necessary and useful to merge different data structures to one and the same block RAM.

Solution ⑤ for instance requires only 30 mW more power but it uses two block RAMs less compared to ①. In this case the Deque of Module B is mapped to the first port of a dual port block RAM and the Vector and the Queue of Module C are mapped to the second port. The List is mapped to a different block RAM since its bit width is different compared to the other containers. The third block RAM is used as ROM for the Const Array of Module B.

The solutions ②, ③ and ④ represent different other constellations, where some containers are realized using block RAM and some are realized using distributed RAM.

VI. Conclusion and Outlook

In this paper the Hardware/Software Codesign Template Library has been presented. Together with the Tripartite system level design flow, the design space exploration of modern cyber-physical systems is simplified. The possibility to use complex data structures significantly increases the level of abstraction of the system level model. At the same time, the separation of container interface and implementation allows a simple mapping to either hardware or software. Additionally, by the memory mapping feature different memory structures in hardware are considered and an efficient usage of these structures is enabled.

The case study shows the applicability and how simple high level algorithms can be mapped to different hardware structures. The resulting estimates of the required resources and the power consumption help the designer to find the best design solution at the system level.

For the future as well an extension of container types as an extension of the library's capabilities would be possible. Currently, a limiting factor is the restriction of the HLS tools. By a better support of pointers, an extension of the library towards concepts like iterators and functors would be conceivable.

References

[1] P. Brunmayr, J. Haase, and C. Grimm, "A tripartite system level design approach for design space exploration," in *Proc. of the 2010 Forum on specification & Design Languages*, September 2010, pp. 50 – 55.

[2] SystemC™. [Online]. Available: http://www.systemc.org

[3] Cynthesizer Data Sheet. ForteDS. [Online]. Available: http://www.forteds.com/products/cynthesizer_datasheet_2008.pdf

[4] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, April 2003.

[5] 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 Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, August 2007.

[6] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, pp. 542–555, 2008.

[7] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski, "System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design," *EURASIP Journal on Embedded Systems*, vol. 2008, p. 13, 2008.

[8] SpecC™. Website. SpecC Technology Open Consortium. [Online]. Available: http://www.specc.org

[9] S. Park, S. Yoon, and S. Chae, "A mixed-level virtual prototyping environment for refinement-based design environment," in *7th IEEE International Workshop on Rapid System Prototyping*, 2006, pp. 63–68.

[10] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith, "Systemcodesigner: Automatic design space exploration and rapid prototyping from behavioral models," in *Proceedings of the Design Automation Conference*, 2008.

[11] L. Pomante, "Experimenting Object-Oriented System-Level Design in the ATM domain," in *IEEE/ACM/IFIP Intl. Conf. on Hardware/software codesign and system synthesis*, 2004.

[12] B. Liskov and S. Zilles, "Programming with abstract data types," in *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, 1974.

[13] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

[14] Xilinx. Virtex-4 Family Overview. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[15] Synplify premier data sheet. Synopsys. [Online]. Available: http://www.synopsys.com/Tools/Implementation/FPGAImplementation/CapsuleModule/syn_prem_ds.pdf