System Level Design Using C++

Diederik Verkest, IMEC, Leuven, Belgium, verkest@imec.be
Joachim Kunkel, Synopsys, Mountain View, CA, kunkel@synopsys.com
Frank Schirrmeister, Cadence Design Systems, San Jose, CA, franks@cadence.com

Abstract

This paper discusses the use of C++ for the design of digital systems. The paper distinguishes a number of different approaches towards the use of programming languages for digital system design and will discuss in more detail how C++ can be used for system modeling and refinement, for simulation, and for architecture design.

1. Introduction

Advances in silicon processing technology enable integration of ever more complex systems on a single chip. These so-called Systems-on-Chip (SoC) contain dedicated hardware components, programmable processors, memories, … requiring not only the design of digital hardware but also the design of embedded software. The successful deployment of these systems requires a very high design productivity to deal with the immense complexity of the system with limited design resources. This design productivity problem is at the basis of the recent paradigm shifts in system design: Intellectual Property (IP) reuse, platform based-design, and high abstraction level specifications covering both hardware and embedded software aspects of a SoC.

A higher abstraction level entry to the SoC design process can be achieved by extending a software programming language such as C/C++ (see further) or JAVA (e.g. [13,18,20,36]), using a specialized language such as SDL (e.g. [14,35]), or by extending an existing hardware description language to deal with system-level concepts (e.g. [1]). In this paper we discuss the use of the C++ programming language to design SoCs from the system-level down-to implementation.

In today's system level design flows it has become commonplace to describe the pure functional system level model of hardware building blocks in C/C++. Once hardware-related concepts need to be expressed though, the design usually is transferred into a hardware description language (HDL) based design environment. The reason for this is that C/C++ on and by itself does not provide the necessary mechanism to describe concepts like concurrency, signals, reactivity, hardware data-types, … that are inherent to hardware. The problem with this approach is that it requires rewriting the C/C++ description in HDL, a time consuming and error prone step, and even worse, it usually also forces the handoff of the design from a C++ knowledgeable systems engineer to an HDL trained ASIC designer. Needless to say that once the design has been converted to HDL, the system engineer's ability to continue contributing to the data and control refinement process vanishes.

Originally intended for software design, neither C nor C++ has the basic support required for accurately dealing with the hardware parts of a SoC design. There are two approaches for building this type of hardware support. The first approach relies on *syntax extension* [2,7,9,12,16] and requires the development of separate compilers, simulators, and synthesis tools to manipulate the new syntax. In this paper we look at the second approach which relies on *class libraries* to model the hardware aspects of the SoC design and can only be used with languages that are extensible such as C++ or JAVA. These approaches can re-use the existing language development framework (compilers, debuggers, and other productivity tools), for instance for simulation.

There are several ways of using C++ in the SoC design process. The most straightforward use of C++ is to construct a pure functional executable model for simulation using the C++ language and additional class libraries for hardware concepts. A more sophisticated use of C++ makes use of the class libraries and the C++ language framework to support a systematic methodology to design and integrate parts of the SoC [19,23,25]. Finally, a properly defined subset of the C++ language and the class library extensions can be used as the input to hardware synthesis [10].

In section 2 we discuss the use of C++ for modeling systems at different abstraction levels and pay attention to the mechanisms that C++ offers for support of IP reuse. In section 3 we discuss the use of C++ for simulation and architecture definition. Conclusions are due in section 4.

2. System modeling and refinement

The object-oriented and multi-paradigm modeling capabilities of C++ [4,29] make it an attractive language for system-level design. C++ mechanisms such as classes, templates and operator overloading can be used to elegantly integrate features that are not available in basic C++. Classes, for instance, can be used to define special bit-true data-types [15,17,33]. Operators can be overloaded to refine their semantics, for instance to model the overflow behavior of operators on bit-true data-types or to add execution times for performance modeling. This extensibility is crucial to support a mix of modeling paradigms in a single language framework and makes C++ well suited for executable system modeling. Using C++, it is possible to provide a rich set of modeling primitives to system designers without creating a new syntax or compiler [6,10,11,23,32,34,37].

We believe system-level design should be based on an executable model in which functionality, architecture, and timing can be refined concurrently from concept to implementation. To be useful for system-level SoC design, this executable model has to support the different computational models (communicating concurrent processes, data-flow, finite-state machines, discrete event, etc.) that are required to deal with the specification heterogeneity of SoC.

The key to mastering the complexity of SoC design is abstraction. In the next sections we review two approaches that apply this principle to support SoC design.

2.1. A C++ class library based approach

While the key to mastering the complexity of SoC design is to move to higher levels of abstraction, it is also important to continue supporting abstraction levels and concepts well established in the design community. A good example for this approach is SystemC [40].

SystemC provides a C++ class library based implementation of objects like processes, ports, signals, hardware data types, ... as well as an event based simulation kernel. Using the SystemC C++ class libraries and an off the shelf C++ compiler, a designer can describe hardware components at a broad range of abstraction levels, which result from the ability to perform data refinement and control refinement separately. One extreme example is C++ code that does not include any SystemC objects other than the hardware data types, like bit vectors or finite precision signed and unsigned integers. An extreme example at the other end of the spectrum is a functional C++ description exhibiting clock cycle accurate behavior only at the ports of the hardware component.

SystemC traces back its origin to the Scenic [10] project at Synopsys. One of the goals of the Scenic project was to create a C++ based hardware description language that could be used to create executable specifications of hardware components and would also allow for RTL and behavioral level synthesis.

The following is an example of a 16-bit CRC generator described using the class libraries that are available in the SystemC environment (some modeling constructs are highlighted in bold).

```
struct crc_ccitt : sc_module {
      sc_in<bool> in;
      sc_out<sc_bv<8>> out;

      SC_CTOR(crc_ccitt)
      {
        SC_CTHREAD(entry);
        watching(reset.delayed() == 1);
      }

      void entry();
};

void crc_ccitt::entry()
{
      bool s;
      sc_bv<16> crc;

      if (reset == 1) {
            out = 0;
      }
      wait();

      while (true) {
            crc = out;
            s = crc[15] ^ in;
            out = (crc.range(14,12),
                   crc[11] ^ s,
                   crc.range(10,5),
                   crc[4] ^ s,
                   crc.range(3,0), s);
            wait();
      }
}
```

Given its well defined HDL-like RTL and behavioral level hardware semantics, SystemC provides a very valuable language foundation for the development of C++ based synthesis tools that leverage today's proven HDL based synthesis technologies. Since SystemC follows the C++ class library based approach, introducing additional system design concepts merely requires the creation of new class libraries.

2.2. An object-oriented approach

Further abstraction can be obtained by capturing the intended design concepts in C++ objects and offering these as class libraries to the system designer. At the synthesizable RT level, for example, designers reason with the synchronous FSMD paradigm as the key design

concept [8] and not with event-driven semantics of a traditional HDL. Similarly, DSP systems are specified at the system-level using a data-flow paradigm. So in addition to supporting event-driven simulation semantics in C++, higher conceptual abstraction levels such as FSMD and data-flow objects [23], concurrent process models and time [30], ... are required to construct high-level models of SoC designs. These more abstract SoC models allow a designer to effectively explore the design space to an extent that is no longer possible at the detailed RT HDL level.

Such a high-level executable system model is only the starting point. To make the use of C++ for SoC design really useful, a design environment is required in which the executable model can be refined towards an implementation. A proper use of the C++ language framework supports this refinement while limiting the amount of code rewriting a designer has to carry out. SoC design can then become a programming activity where the link from the system-level specification to the RT level implementation is carried out entirely inside the C++ environment using a process of incremental refinement. This gradual introduction of implementation detail in the executable specification of the SoC requires the ability to combine the different modeling objects in one and the same description [23]. Part of the SoC design can be at e.g. the abstract data-flow level and other parts can already be at the more refined FSMD level. This incremental refinement can be carried out under the control of the designer where important decisions are left to the designer. The tools give the required feedback that allow the designer to make the right decisions and support the designer in changing the SoC model to reflect the implementation decisions. This type of incremental refinement flow using the C++ framework has been applied in several application domains, such as network protocol processing (ATM) [5,31] and digital telecom systems [3,22,23,24] including embedded software components [6]. The next paragraphs illustrate how these principles have been applied in the OCAPI design environment [23,38].

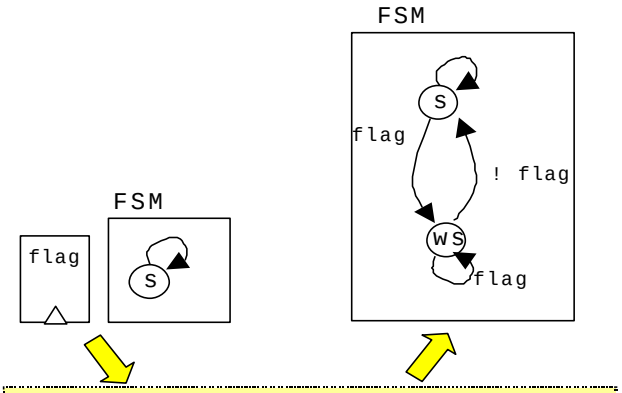
```
Architecture RTL of my_processor is
begin
  SYNC: process (clk)
  begin
    if (clk'event and clk = '1') then
      current <= next_state;
      a_at1 <= a;
    end if;
  end process;

  COMB: process
  begin
    a <= a_at1;
    case current is
    when state1 =>
      a = 0;
      next_state <= state2;
    when state2 =>
      a = a_at1 + 1;
    end case;
    if (reset = '0') then
      a = 0;
    end if;
  end process;
end RTL;
```

```
#include "ocapi.h"

void main() {
  sig a (ck);     // register

  sfg reset;      // instruction
  a = 0;

  sfg inc;
  a = a +1;

  fsm f(ck);      // controller
  state state1, state2;

  f << deflt(state1);
  f << state2;

// state transitions
state1 << always << reset << state2;
state2 << always << inc << state2;
}
```

The examples above show VHDL and C++ descriptions for a simple increment block implemented as a synchronous digital machine with one controller sending instructions to a data path. In the VHDL description it is not possible to clearly distinguish the code used for modeling the controller and the code used for modeling the data path. In addition, the VHDL constructs used have no direct relation to the RT level architecture of the increment block. The C++ description on the contrary, uses objects such as **`sfg`**, **`state`**, and **`fsm`** to reflect the exact design concept that was intended. The **`sfg`** object models a data path instruction, while **`state`** and **`fsm`** are parts of the controller. All these objects are related to each other through the use of (overloaded) C++ operators and expressions. As these operators execute, an object hierarchy is constructed that reflects the RT behavior of the processor. The object hierarchy can be manipulated for simulation, code generation, or other purposes. In contrast to a traditional use of a programming language to directly express the functional behavior of the design, OCAPI descriptions express a conceptual model of the design. A traditional C++ description of the increment block would, upon execution, produce a sequence of incrementing numbers. The OCAPI description, upon execution, produces a model of the increment block that can subsequently be manipulated e.g. interpreting it to produce the sequence of incrementing numbers, or generating equivalent HDL code from it that can be used for synthesis. It is this ability to manipulate the design

description in the same C++ environment that also enables effective IP re-use, as will be discussed in the next paragraph.

IP reuse is in the first place a matter of reusing functionality, not structure. Structural reuse forces the designer to reuse the implementation of an IP block "as is". However, an optimal integration of an IP block in a different system context often requires changes, which are nearly impossible to carry out on a structural or even RT VHDL level description of the IP block where communication, control, and functional aspects of an IP block are all intermixed. Hence, one of the keys to reuse is to provide a clear separation between the functionality (i.e. internal behavior) of an IP block and its communication (i.e. link with the external world) [2,26,27]. This principle leads to a design methodology known as interface-based design [17,21,42] where the interface of a component can be refined independently from the component's internal behavior. Although interface-based design is an important step forward to enable effective IP integration, it leads to separately refined communication and functional behavior that can not, afterwards, be integrated easily. In the object-oriented design approach embedded in the OCAPI environment, the principle is taken one step further leading to behavioral re-use [24] where the re-use objects can directly access the *internal* basic *behavioral* objects of the components of the system description: `STATE`, `FSM`, `SFG`, etc. In the OCAPI environment one can implement methods that not only refine the communication but also allow the refined communication to automatically manipulate the IP block's behavior and adapt it to the new communication environment. This manipulation is possible because executing an OCAPI description of a system creates an object hierarchy (i.e. a data structure) of the system, built-up from OCAPI's basic behavioral objects, rather than directly executing the system behavior. This data structure can be manipulated by having other objects interact with it to add, for example, new states and transitions to the FSM object.




```
class waitstate {
  state ws;
  public:
    waitstate();
    expand (fsm &f, sig &flag, state &s) {
      f << ws;
      s << cnd(flag) << ws;
      ws << cnd(flag) << ws;
      ws << !cnd(flag) << s;
    }
};
```

Figure 1. Manipulation of design objects

The example in Figure 1 shows how a `waitstate` object attaches itself to an existing FSM. Given the FSM `f`, a start state `s`, and a signal `flag`, it modifies the FSM such that it contains a conditional jump from the start state to a newly created wait state (`ws`). The procedure that performs the modification is the `expand` method that is given the necessary hooks (FSM `f`, state `s`, and signal `flag`) to attach to. This type of reuse allows the introduction of very complex communication behavior in a system. The designer is responsible only for the description of the data processing parts of the system and does not need to worry about the interaction with e.g. an I^2C programming interface. The programming interface protocol is available as a reuse object in the OCAPI environment and knows how to attach itself to the data processing part of the system and modify the control-flow of the data processing part to provide an optimal integration.

3. Simulation and architecture definition

As pointed out in section 1, there are different levels of abstraction at which C++ can be used for system modeling. Traditional system designers use *pure functional* C++ to model and explore the algorithmic aspects of a block to be designed. At this level of abstraction the designer thinks in terms of abstract tokens like ATM cells and GSM frames. Simulation assumes ideal constraints like zero execution time and infinite queues between data-flow modules, which avoids token loss and would use floating point data-types for the arithmetic. As the result of this step the designer understands whether an algorithm is the right one to choose taking into account, for example, certain channel characteristics in a 3G wireless system. There are really no architectural effects simulated at this level, only the structure of the algorithm and its function.

When implementing a particular block into hardware or software the first architectural effect to be analyzed is the width of data used for the algorithmic part of the design. In a simulation the arithmetic operators can be overloaded in C++ to work on implementation data-types. A polymorphic type system, which supports the

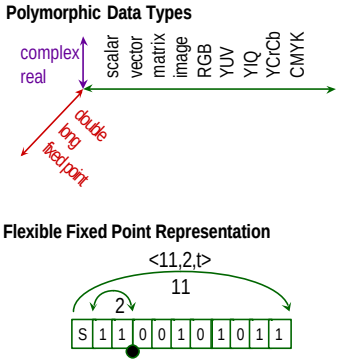


Figure 2. Polymorphic data-types

appropriate fixed point data-types (Figure 2), is an enabling factor for seamless refinement between different arithmetic data-types. It allows (together with operator overloading in C++) to change the data-types operated on without having to recapture the structure of the design. At the end of this step the designer understands which effect the architectural limitation of the data width (defined in the type system) has on the algorithm's functionality, and whether e.g. truncation, rounding or error correction algorithms have to be inserted.

After the data width has been analyzed and decided, the implementation proceeds with adding the effects of fine grain architectural effects like pipelines and register banks. The designer may use C++ combined with appropriate class-libraries to model effects like concurrency and parallelism in a block to be designed. The architectural effects are, therefore, inserted using the class library extensions. The C++ code represents essentially a hardware description of the block (and also looks very similar to HDL representations of the same block). The step from this model to a RTL description becomes mainly a translation step as they work on the same level of design abstraction.

The techniques described above for C++ based simulation and architecture definition work very well for the actual implementation of a block and the refinement from an abstract algorithm down to RTL implementations. Figure 3 summarizes the different steps of the *IP Authoring Flow* described above.

Taking into account the exploding complexity of today's SoC designs it is important to note that there is a second, very important flow. It is *IP Block Integration.* This second flow enables the exploration of block functionality in its system context at the abstract system level (C++ untimed, performance), and allows the refinement of an executable system specification from tokens to the actual signal implementations.

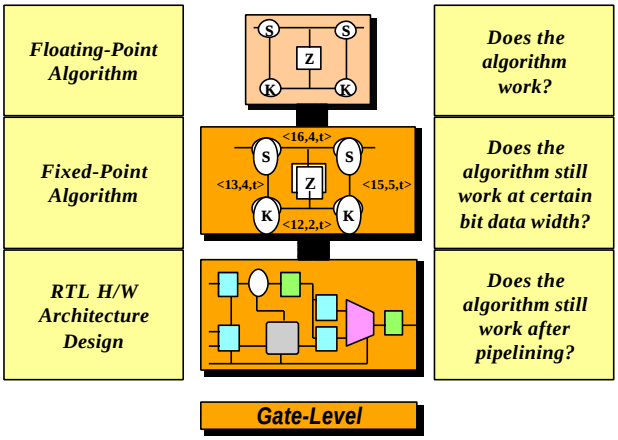


Figure 3. Levels of abstraction for IP authoring

Starting from the abstract untimed pure functional system level model described above, design teams have to understand how a particular block interacts with its environment from the algorithmic perspective. An example would be a physical layer channel receiver, error correction and speech decoding arranged in a serial fashion in a wireless receiver system. In an IP Reuse scenario these models might come from different sources, potentially even as black boxes without access to the module content. Furthermore, at this level of functional integration, different models of computation like data flow, discrete event, continuous time, ... have to be integrated into one simulation model, raising several issues related to simulation (see for more details below).

While the designer does analysis at this level within blocks, e.g. bit error rate analysis, a typical integration effect to be investigated would be whether the error correction algorithm can recover dropped symbols from the physical layer decoder. The analysis of the integration is at this point fully functional, no architectural effects are considered, and only the structure of the algorithm is taken into account.

From an *IP Integration* perspective C++ is probably only one potential authoring technique, since several best in class description technologies have been established in the recent past. Among them most notably UML-RT and SDL are used for software modeling, while specialized description techniques like data flow networks (as used in Cadence Cierto™ SPW, Synopsys COSSAP™) or emerging languages like Superlog™ [41] are point description techniques for hardware modeling. Therefore it is essential to allow efficient exchange of simulation models between simulators. The IEEE Open Model Interface (OMI) standard (see also [39]) allows this exchange of protected simulation models between simulators. Wrapped with an OMI compliant interface, modules can be linked into C++ based simulations.

After the designer has ensured correct integration of algorithms from different sources at the untimed abstract level, the next step is to analyze the performance impact of the architectural partitioning. It is desirable to refine the pure functional C++ based (or OMI imported) simulation models with additional performance information. This information is typically associated with coarse grain architectural components like CPUs, DSPs, bus systems, memories, arbiters and schedulers in real time operating systems. A function independent characterization of performance can be achieved by providing C++ based APIs for accessing scheduler or arbitration models. These APIs then represent the C++ based architectural models, which carry the performance and delay in which the pure functional models execute or exchange information among each other.

By adding performance aspects during the integration of IP blocks, important integration aspects can be gauged very early in the design cycle prior to implementation. While there are several aspects to be considered especially between the different models of computation (data flow, discrete event, continuous time, etc.), let us pick the example of data flow integration into a control oriented discrete event environment.

In the untimed abstract data flow simulation a schedule typically is defined which calls the different data flow elements. Infinite queues are assumed between the different modules at this level, which means that tokens are never lost. When integrated in a discrete event simulation environment and after performance characterizations are attached, such data flow simulation can now take into account the reality that infinite queues do not exist. The physical layer receiver in the example above might produce tokens in a different rate than the speech decoder can consume them. The implementation of the speech decoder might be too slow, therefore the simulation has to detect that queues are overflowing and eventually tokens will be lost if non sufficient queue control is applied.

Finally, the design has to be refined from the untimed performance level of abstraction to the fully timed, clocked level from which the design can be exported to the actual hardware and software implementations. The abstract tokens used at the untimed system level (GSM frames, ATM cells) can be refined into actual signal transactions by adding the appropriate handshake detail in a communication refinement step. Inter module communication patterns represent the path that a signal takes through the architecture (e.g. an interrupt transmitted over a bus to a CPU triggering an interrupt service routine receiving data using shared memory).

The different levels of abstraction in an IP integration process are indicated in Figure 4. For further information about the methodology of system level integration and successive refinement to implementation see [19,25].

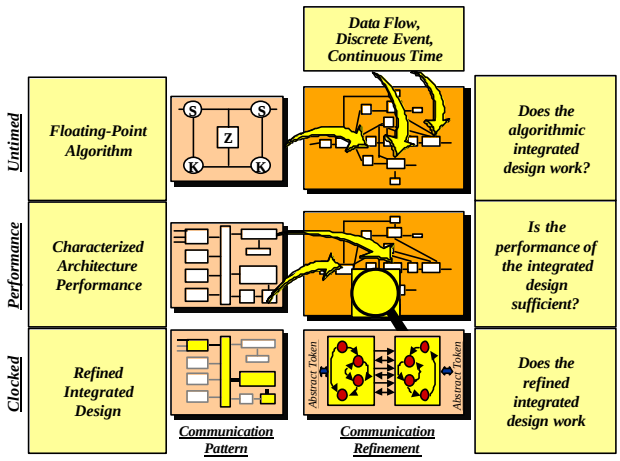


Figure 4. Levels of abstraction for IP integration

4. Conclusions

The object-oriented and multi-paradigm nature of the C++ language allow the introduction of typical hardware design concepts such as bit-true data-types, finite state machine models, ... which extend the basic semantics of C++ towards system-level design. In this paper we have discussed the use of these principles for system design modeling at the RT, architecture, and system level and for the refinement between these different abstraction levels. The advantages of C++ for system-level design were reviewed. By making extensive use of class libraries, C++ offers a single language framework for executable system modeling and incremental refinement towards an implementation. In case of object-oriented modeling with objects that directly correspond with synthesizable concepts, as explained in section 2.2, the lowest refinement level will be an RT-level C++ description from which proven synthesis techniques can be used to generate gate-level implementations.

Synthesis of the full C/C++ language is mostly similar to synthesis from HDL descriptions, which is a mature technology today. However, C/C++ presents some unique challenges for synthesis. Firstly, synthesis of descriptions using pointers is difficult. Due to effects such as aliasing, analysis of pointers and where they point to is non-trivial. Though one can restrict synthesizable descriptions to exclude pointers, this problem cannot be ignored in the long term. While some progress has already been reported in this area [28,31], more work needs to be done. Another challenge is in extending the synthesizable subset to include object-oriented features like virtual functions, multiple inheritance, etc. More work is required to define the object-oriented semantics for hardware before synthesis can be attempted.

Finally, a proper use of object-orientation supports a much improved IP re-use process both for IP authoring and IP integration.

Acknowledgements

The authors would like to acknowledge the contributions of many colleagues: Patrick Schaumont, Johan Cockx, Chantal Ykman, Dirk Desmet from IMEC, Abhijit Ghosh, Stan Liao from Synopsys, and Stan Krolikoski from Cadence. Further thanks go to Luciano Lavagno for his valuable comments on earlier drafts.

IMEC acknowledges the financial support of the Flemish IWT and Alcatel in the context of the MEDEA A-403 SMT, MEDEA A-114 xDSL, and ITA-2 IRMUT projects and of the European Commission in the context of the EP 21,929 MEDIA project.

References

[1] P. Ashenden and M. Radetzki, "Comparison of SUAVE and Objective VHDL Language Features, *Proceedings of FDL-99*, pp. 269-278, Lyon, France, September 1999.

[2] I. Bolsens, H. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest, "Hardware-Software Co-Design of Digital Telecommunication Systems", *Proceedings of the IEEE*, 85(3):391-418, March 1997.

[3] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens, "A Methodology and Design Environment for DSP ASIC Fixed Point Refinement", *Proceedings of DATE-99*, pp. 271-276, Munich, Germany, March 1999.

[4] J. Coplien, "Multi-Paradigm Design for C++", *Addison-Wesley*, October 1998.

[5] J. da Silva, C. Ykman-Couvreur, M. Miranda, K. Croes, S. Wuytack, G. de Jong, F. Catthoor, D. Verkest, P. Six, and H. De Man, "Efficient System Exploration and Synthesis of Applications with Dynamic Data Storage and Intensive Data Transfer", *Proceedings of the 35th DAC*, pp. 76-81, San Francisco, CA, June 1998.

[6] D. Desmet, M. Esvelt, P. Avasare, D. Verkest, and H. De Man, "Timed Executable System Specification of an ADSL Modem Using a C++ Based Design Environment", *Proceedings of CODES-99*, pp. 38-42, Rome, Italy, May 1999.

[7] R. Ernst, J. Henkel, and T. Benner, "Hardware/Software Cosynthesis for Microcontrollers, *IEEE Design and Test of Computers*, pp. 64-75, December 1993.

[8] D. Gajski, N. Dutt, A. Wu, and S. Lin, "High Level Synthesis", *Kluwer Academic Publishers*, 1992.

[9] D. Gajski, R. Dömer, and Jianwen Zhu, "IP-centric Methodology and Design with the SpecC Language", *Proceeding of the NATO ASI on System Level Synthesis for Electronic Design*, Chapter 10, Il Ciocco, Lucca, Italy, August 1998. Edited by A. Jerraya and J. Mermet, Kluwer Academic Publishers, May 1999.

[10] A. Ghosh, J. Kunkel, and S. Liao, "Hardware Synthesis from C/C++", *Proceedings of DATE-99*, pp. 387-389, Munich, Germany, March 1999.

[11] R. Gupta and S. Liao, "Using a Programming Language for Digital System Design", *IEEE Design and Test of Computers*, pp. 72-80, April-June 1997.

[12] R. Gupta and G. De Micheli, "Hardware/Software Cosynthesis of Digital Systems", *IEEE Design and Test of Computers*, pp. 29- 41, September 1993.

[13] R. Helaihel and K. Olukotun, "JAVA as a Specification Language for Hardware/Software Systems", *Proceedings of ICCAD-97*, pp. 690-697, San Jose, CA, November 1997.

[14] A. Jerraya and K. O'Brien, "SOLAR: An Intermediate Format for System-Level Modeling and Synthesis", in *Computer Aided Software/Hardware Engineering*, J. Rozenblit, K. Buchenrieder, eds, IEEE Press, 1994.

[15] S. Kim, K. Kum, and W. Sung, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs", *Workshop on VLSI Signal Processing*, pp.197-206, Osaka, Japan, November 1995.

[16] L. Lavagno and E. Sentovich, "ECL: A Specification Environment for System-Level Design", *Proceedings of 36th DAC*, pp. 511-516, New Orleans, LA, June 1999.

[17] C. Lennard, P. Schaumont, G. de Jong, and P. Hardee, "Standards for System-Level Design: Practical Reality or Solution in search of a question?", *Proceedings of DATE-2000*, Paris, France, March 2000.

[18] O. Levia and C. Ussery, "Directed Control Data-flow Networks: A New Semantic Model for the System-on-Chip Era", *Proceedings of FDL-99*, pp. 548-560, Lyon, France, September 1999.

[19] G. Martin and Sanjay Chakravarty, "A New Embedded System Design Flow based on IP Integration", *Proceedings of DATE-99 User Forum*, pp. 99-103, Munich, Germany, March 1999.

[20] C. Passerone, C. Sansoè, L. Lavagno, R. McGeer, J. Martin, R. Passerone, and A. Sangiovanni-Vincentelli, "Modeling Reactive Systems in JAVA", *ACM Transactions of Design Automation for Electronic Systems*, 3(4):515-523, October 1998.

[21] J. Rowson and A. Sangiovanni-Vincentelli, "Interface-Based Design", *Proceedings of the 34th DAC*, pp. 178-183, Anaheim, CA, June 1997.

[22] P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens, "Synthesis of Multi-Rate and Variable Rate Digital Circuits for High Throughput Telecom Applications", *Proceedings of ED&TC-97*, pp. 542-546, Paris, France, March 1997.

[23] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens, "A Programming Environment for the Design of

Complex High Speed ASICs", *Proceedings of the 35th DAC*, pp. 315-320, San Francisco, CA, June 1998.

[24] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, and I. Bolsens, "Hardware Reuse at the Behavioral Level", *Proceedings of the 36th DAC*, pp. 784-789, New Orleans, LA, June 1999.

[25] F. Schirrmeister and S. Krolikoski, "Virtual Component Co-Design – Facilitating a Win-Win Relationship between IP Integrators and IP Providers", *Fall IP Conference 1999*, Edinburgh, 1999.

[26] C. Schneider and W. Ecker, "Stepwise Refinement of Behavioral VHDL Specifications by Separation of Synchronization and Functionality", *Proceedings of EURODAC-96*, pp. 509-514, Geneva, Switzerland, September 1996.

[27] G. Schumacher, W. Nebel, and C. von Ossietzky, "Object-Oriented Modeling of Parallel Hardware Systems", *Proceedings of DATE-98*, pp. 234-241, Paris, France, March 1998.

[28] L. Semeria and G. De Micheli, "SpC: Synthesis of Pointers in C. Application of Pointer Analysis to the Behavioral Synthesis from C", *Proceedings of ICCAD-98*, pp. 340-346, November 1998.

[29] B. Stroustrup and S. Hamilton, "The Real Stroustrup Interview", *Computer*, pp. 110-114, June 1998.

[30] D. Verkest, J. Cockx, F. Potargent, G. de Jong, and H. De Man, "On the Use of C++ for System-on-Chip Design", *Proceedings of IEEE CS workshop on VLSI-99*, pp. 42-47, Orlando, FL, April 1999.

[31] D. Verkest, J. da Silva, C. Ykman, K. Croes, M. Miranda, S. Wuytack, F. Catthoor, G. de Jong, and H. De Man, "Matisse: A System-on-Chip Design Methodology Emphasizing Dynamic Memory Management", *Journal of VLSI Signal Processing*, 21(3):185-194, July 1999.

[32] C. Weiler, U. Kebschull, and W. Rosenstiel, "C++ Base Classes for Specification, Simulation and Partitioning of Hardware/Software Systems", *Proceedings of ASP-DAC*, CHDL, VLSI, pp. 777-784, 1995.

[33] M. Willems, V. Bursgens, H. Keding, T. Grötker, and H. Meyr, "System Level Fixed-Point Design Based on an Interpolative Approach", *Proceedings of the 34th DAC*, pp. 293-298, Anaheim, CA, June 1997.

[34] J.-S. Yim, Y.-H. Hwang, C.-J. Park, H. Choi, W.-S. Yang, H.-S. Oh, I.-C. Park, and C.-M. Kyung, "A C-based RTL Design Verification Methodology for Complex Microprocessors", *Proceedings of the 34th DAC*, Anaheim, CA, June 1997.

[35] C. Ykman-Couvreur, J. Lambrecht, D. Verkest, B. Svantesson, A. Hemani, S. Kumar, and F. Wolf, "System Exploration and Synthesis from SDL of an ATM Switch Component", *Proceedings of 12th ASIC/SOC Conference*, pp. 119-124, Washington, DC, September 1999.

[36] J. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and R. Newton, "Design and Specification of Embedded Systems in JAVA Using Successive, Formal Refinement", *Proceedings of the 35th DAC*, pp. 70-75, San Francisco, CA, June 1998.

[37] Cynlib, http://www.CynApps.com/.

[38] OCAPI, http://www.imec.be/ocapi/.

[39] Open Model Interface Standard, see http://www.cfi.org/OMF/.

[40] Open SystemC Initiative, http://www.SystemC.org/.

[41] Superlog, http://www.co-design.com/.

[42] VSIA System-Level Interface Behavioral Documentation Standard, System Level Design Development Working Group Standard 1 Version 0.1, http://www.vsi.org/.