

# A Methodology to Recover RTL IP Functionality for Automatic Generation of SW Applications

Nicola Bombieri, University of Verona  
Franco Fummi, University of Verona  
Sara Vinco, University of Verona

With the advent of heterogeneous multi-processor system-on-chips (MPSoCs), hardware/software partitioning is again on the rise both in research and in product development. In this new scenario, implementing intellectual-property (IP) blocks as SW applications rather than dedicated HW is an increasing trend to fully exploit the computation power provided by the MPSoC CPUs. On the other hand, whole libraries of IP blocks are available as RTL descriptions, most of them without a corresponding high-level SW implementation. In this context, this article presents a methodology to automatically generate SW applications in C++, by starting from existing RTL IPs implemented in hardware description language (HDL). The methodology exploits an abstraction algorithm to eliminate implementation details typical of HW descriptions (such as, cycle-accurate functionality and data types) to guarantee relevant performance of the generated code. The experimental results show that, in many cases, the C++ code automatically generated in few seconds with the proposed methodology is as efficient as the corresponding code manually implemented from scratch.

Categories and Subject Descriptors: C.3 [SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS]

General Terms: Design, Performance

Additional Key Words and Phrases: RTL IP, IP reuse, Embedded Software Generation

## ACM Reference Format:

Nicola Bombieri, Franco Fummi, and Sara Vinco, 2014. A Methodology to Recover RTL IP Functionality for Automatic Generation of SW Applications. *ACM TODAES* V, N, Article A (January YYYY), 25 pages.  
DOI: <http://dx.doi.org/10.1145/0000000.0000000>

## 1. INTRODUCTION

Heterogeneous multi-processor systems-on-chips (MPSoCs) are increasingly being used in both embedded and low-end general purpose systems to overcome poor performance scalability and energy efficiency of single processor systems-on-chips (SoCs) [Wolf et al. 2008].

The rising complexity of MPSoCs, which incorporate several programmable devices (e.g., general purpose processors, digital signal processors, application specific instruction set processors) requires software and hardware designers together with system architects to take into account new aspects for HW/SW partitioning [Martin 2006]. In fact, in this new context, HW/SW partitioning is influenced by the high computing

---

Extension of Conference Paper: This article starts from the RTL abstraction technique presented in [Bombieri et al. 2010]. The added novel contributions are listed at the end of the related work section.

This work has been partially supported by the European Project TOUCHMORE FP7-ICT-2011-7-288166.

Author's addresses: Nicola Bombieri and Franco Fummi and Sara Vinco, Computer Science Department, University of Verona.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© YYYY ACM /YYYY/01-ARTA \$15.00

DOI: <http://dx.doi.org/10.1145/0000000.0000000>

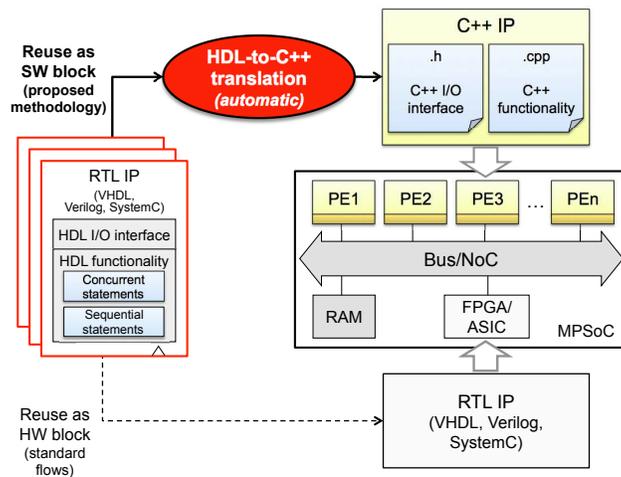


Fig. 1. HW-to-SW migration of existing RTL IPs in modern HW/SW MPSoC partitioning

power of the multiple processing elements (PEs) throughout the MPSoC design space exploration. On the one hand, the hardware accelerators implementing any IP functionality is two orders of magnitude faster than any corresponding C++ applications running on a general purpose CPU. On the other hand, hardware accelerators (such as FPGAs) are expensive and limited in MPSoCs, while there is no limit to the number of IP functionality running on the several processors. For this reason as well as for the maintainability advantages of the SW components, more and more system functionality is defined to become SW applications rather than dedicated hardware blocks (*HW-to-SW migration*) [ITRS 2011; Freescale 2009].

Reuse of existing and already verified RTL IP components is a key strategy to cope with the complexity of designing modern SoCs under ever stringent time-to-market requirements. To achieve a 10x gain in design productivity by the year 2020 is expected to require that a complex SoC will consist of 90% reused components [ITRS 2011]. In this context, while new SoC designs are increasingly started at the system level, there is a large body of RTL IP blocks which have been designed in VHDL or Verilog by both industry designers and third-party vendors over the years. In standard flows, reusability of such RTL IPs in hardware accelerators (see lower side of Figure 1) is not always guaranteed since it depends on the designers' ability to implement them independently from a specific integration context. Still, their reuse often requires a manual, time consuming end error prone customization which often eludes the advantages of the IP reuse.

The motivation for this work is precisely the fact that it would be nice to *recover* the core functionality of these designs and to make them suitable as SW applications. This allows designers to use them for system-level design, to avoid their re-implementation from scratch, to enhance their reusability, and, ultimately, to exploit the computational hardware resources (i.e., CPUs) of a MPSoC. This also enables the HW-to-SW migration of IP functionality, which is an increasing trend and requirement in the current and next-generation heterogeneous multicore platforms. As an example, such a migration allows designers to implement cutting-edge policies of *energy and variability awareness* task allocation. Energy and variability awareness is a key issue since it allows performance uncertainties and variations between different chips (die-to-die variations due to aging, temperature, etc.) to be compensated or hidden at the system software level through appropriate workload allocation [Ayad et al. 2013].

The main target of the proposed approach is not RTL IPs that implement commonly used algorithms (e.g., FFT, CRC, etc.), for which designers can usually rely on already existing, tested and optimized C++ implementations. Rather, the approach targets IP blocks like custom IPs (e.g., filters, controllers, etc.) for which designers have the RTL model, but they do not have a corresponding high-level SW implementation. In these cases, such a HW-to-SW migration would require a manual, time consuming and error prone C++ implementation of such models. A similar situation exists when designers have the SW implementation, but there is a disconnection between such high-level specification and the actual RTL hardware. This may occur, for instance, when the RTL has been corrected and refined, but the high-level specification has not.

For the best of our knowledge, reuse of RTL IPs as SW blocks has never been considered so far. Some techniques and tools have been proposed in literature (see the related work section) for translating VHDL/Verilog descriptions into C++ with the aim of improving the performance of HDL model simulations. They propose different optimized approaches to statically solve, at compile time and as much as possible, the process scheduling or to reduce the dynamic scheduling overhead. All these techniques target cycle-accurate simulations to verify the HDL models.

In contrast, the approach proposed in this article aims at generating more abstract C++ implementations, to be executed as SW applications, i.e., whose execution aims at applying the IP functionality over input data and to get the results rather than verifying the correctness of the model itself. Differently from the techniques in literature, the models generated by the proposed methodology are no more cycle accurate and do not need clock accuracy to be simulated. The scheduling semantics of the proposed approach is simulation-like with two important differences: It is not full event-driven since, for instance, it relies on compile-time optimizations like the merge of processes, and it is not a clock semantics since clock cycles are abstracted. Some techniques at the state of the art to semi-statically solve some scheduling activity in the cycle-accurate context are not orthogonal and may be integrated also in our approach. As discussed in the experimental results section, the proposed approach finds the best applicability on custom IPs with simple communication protocols, multi-cycle latency to take advantage from the clock abstraction, and few asynchronous processes to keep the dynamic scheduling overhead limited.

Experimental results also show that, as expected, the performance of the code automatically recovered from RTL descriptions are in general not better than the performance of the code manually implemented. However, the performance are similar in many cases and, for these cases, the proposed approach allows saving manual work for re-implementing and, especially, for verifying SW applications. An analysis is presented to understand for which classes of RTL IPs the methodology is more effective and the characteristics of the RTL descriptions that can influence the performance of the automatically generated C++ code. In HW/SW partitioning, this allows understanding whether there is room for (and thus it is worth) improving the SW application through a manual re-implementation rather than maintaining the IP as HW block.

The article is organized as follows. Section 2 summarizes the related work. Section 3 presents the methodology overview, while the methodology in detail is presented in Sections 4, 5, 6, and 7. Section 8 reports the experimental results and, finally, Section 9 is devoted to concluding remarks and future work.

## 2. RELATED WORK

Several works have been proposed [Hansen 1988; French et al. 1995; Edwards 2003; Saladi et al. 2013] and different commercial tools exist [ALDEC 2014; OSTATIC 2014; DIE.NET 2014; Icarus 2014; VERILATOR 2014; Stoye et al. 2003; Carbon Design Systems 2014] for translating RTL VHDL and Verilog models into C/C++ descriptions,

targeting verification of HW models via simulation. [Hansen 1988] presents data flow analysis to optimize the evaluation of unordered assignment statements that define a hardware structure, and extract clocking rules. [French et al. 1995] proposes an approach to event-driven simulation that does not use a centralized run-time event queue, yet is capable of handling arbitrary models, including those with unlocked feedback and nonunit delay. The elimination of the event queue significantly reduces run-time overhead, resulting in faster simulation. [Edwards 2003] surveys a variety of techniques for translating domain-specific concurrent specifications into sequential code. The techniques address compiling a wide variety of languages, ranging from dataflow to Petri nets. Each uses a different method, to some degree chosen to match the semantics of concurrent language. [Saladi et al. 2013] discusses the application of compiler technology for eliminating redundant computation in hardware simulation. It discusses how concurrency in HDLs presents opportunities for expression reuse across different threads.

[ALDEC 2014] proposes a VHDL-to-C++ converter tool to transform VHDL testbenches to C++ source. During the conversion, the C++ source is compiled into a small simulation kernel that runs the whole simulation with the interconnected hardware board. Tools like [OSTATIC 2014; DIE.NET 2014] allow designers to use C++ executable files in place of VHDL models for decreasing simulation time compared to the typical acceleration process with HDL simulators. All the previous works allow designers to convert VHDL models described by a single process into C++ code, but they do not support scheduling features and synchronization among processes.

In [Stoye et al. 2003], a methodology consequently implemented in the tool VTOC, and in [VERILATOR 2014]), the approaches convert synthesizable Verilog into C++. They perform a synthesis-like transformation of the input Verilog program, resolving the majority of scheduling decisions statically and resulting in a representation of the execution of the Verilog program in each clock cycle. In different ways, the solutions try to reduce the number of delta cycles by topological sorting all processes and by applying process merging. All the implementation details related to HW models (e.g., clock accuracy, bit accuracy) are maintained during the translation to the SW domain. [Icarus 2014] proposes a similar Verilog simulation tool that, differently from [VERILATOR 2014], does not perform code optimization.

Carbon Design System [Carbon Design Systems 2014] provides commercial products that convert Verilog or VHDL RTL models into cycle accurate and register accurate SystemC models. Carbon's tools aim at creating complete virtual platforms in order to gain both a fast and accurate system validation. In contrast, the approach proposed in this article aims at generating fast C++ code to be used as SW application that does not have any detail related to the HW implementation.

[Herrera et al. 2003] presents a different approach, i.e., a method for systematic embedded SW generation that reduces the SW generation cost in a platform-based HW/SW co-design methodology. In particular, C++ code is automatically generated from SystemC processes. Such an approach relies on the overloading of a subset of SystemC constructs. However, it imposes code modification when unsupported SystemC constructs are used in the original description.

Differently from all the techniques presented in literature, the proposed method aims at generating high-level descriptions of IP blocks in C++ to be compiled as SW applications. As discussed in the following sections, the different goal leads to important differences in the approach of generating C++ code.

### 3. METHODOLOGY OVERVIEW

In a HDL model, there are three basic kinds of statements: *declaration* statements, *concurrent* statements and *sequential* statements. Declaration statements are used to

define constants, types, object (i.e., signals, variables, and components) that will be used in the design. Concurrent and sequential statements represent the actual logic of the design and include signal assignments, component instantiations, and behavioral descriptions [VHDL 1994; Verilog 2006; SystemC 2006]. The HDL model consists of:

- An interface  $IF$ , with the sets of input and output ports ( $IN = \{in_1, \dots, in_n\}$ ,  $OUT = \{out_1, \dots, out_m\}$ );
- A set of concurrent statements  $CS = \{cs_1, \dots, cs_k\}$ , which includes the signal assignment statements;
- A set of sequential statements  $SS = \{ss_1, \dots, ss_h\}$ , which includes the statements contained in processes or subprograms;
- A set of synchronous processes  $PS = \{ps_1, \dots, ps_i\}$ , each one containing a set of sequential statements;
- A set of asynchronous processes  $PA = \{pa_1, \dots, pa_j\}$ , each one containing a set of sequential statements.

For the sake of clarity and without loss of generality, components for which the HDL semantics is easily mappable into the C++ semantics (e.g., variables, constants, subprograms, etc.) are not considered in the explanation of the SW generation algorithm. Rather, this article focuses on the most important differences of syntax and semantics between the two languages for modelling IP functionality.

IP functionality are modelled by HDLs through *user-defined processes* (i.e.,  $PS$  and  $PA$ ) that interact each other and with the environment (by means of the interface  $IF$ ). Synchronization among processes is handled by the kernel of the HDL simulator. The simulator kernel coordinates the activity of user-defined processes during a simulation (by means of a *scheduler process*). It also causes the propagation of signal values to occur and causes the values of signals to be updated. Furthermore, it is responsible for detecting events that occur and for causing the appropriate processes to execute in response to those events (by means of an *event queue*) as explained in Section 7.2.

Starting from this HDL model, the proposed methodology consists of four main steps (see Figure 2):

- (1) *Definition of interface and communication protocol.* The C++ model interface and communication protocol are generated from the I/O ports of the HDL model. They are defined by abstracting the pin accurate RTL interfaces and clock accurate communication protocols of the HDL model (Figure 2.1), as described in Section 4.
- (2) *Extended finite state machine (EFSM) generation.* The RTL IP code is parsed and the EFSMs composing the HDL model and representing the IP functionality are extracted (Figure 2.2), as described in Section 5.
- (3) *Merge of processes.* The processes in the HDL model that have the same sensitivity list (i.e., processes that execute in response to the same event/events) are merged into a single process (Figure 2.3). The many processes and concurrent statements that represent the IP functionality in HDL are translated into a high-level SW functionality, which is implemented through fewer functions and variables, as described in Section 6.
- (4) *Mapping of HDL statements into C++ statements.* A set of rules are proposed for translating HDL statements into C++ statements, as explained in Section 7. During translation, the HDL scheduling semantics based on concurrency is abstracted into a more efficient C++ scheduling algorithm and the HDL data types are mapped into efficient C++ bit-accurate data types (Figure 2.4). In addition, a data type abstraction methodology is proposed to abstract the HW dependent details, such as, high impedance, unknown values, etc. of the HDL data types (e.g.,

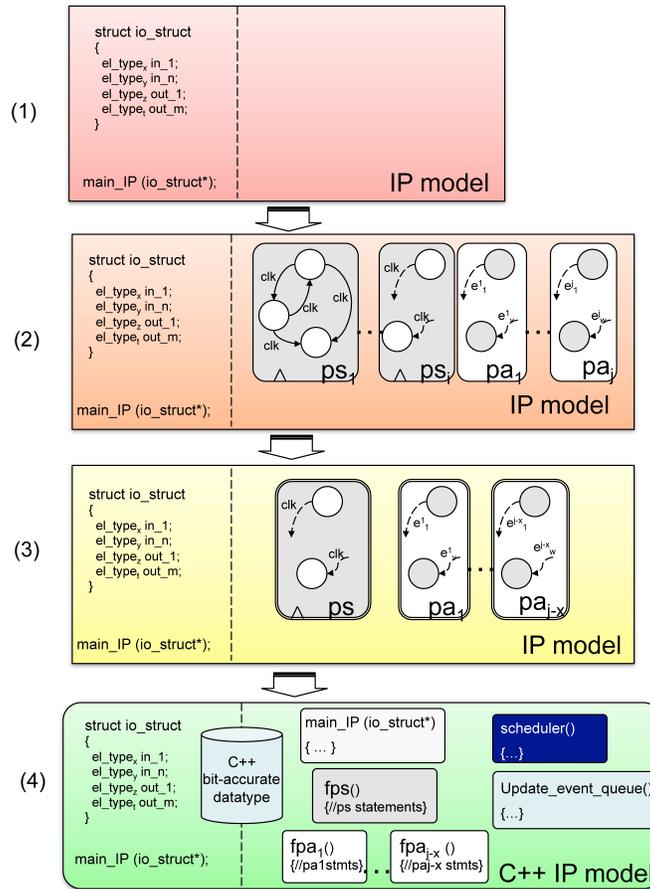


Fig. 2. The C++ generation steps

logic, logic\_vector). The abstraction relies on mapping of the multi-value logic into the two-value logic of the C++ native data types, as described in Section 7.2.

#### 4. DEFINITION OF INTERFACE AND COMMUNICATION PROTOCOL

The *interface* consists of the main IP function (`main_IP()`) and a data structure (`io_struct`), as shown in the left-side of Figure 2(1).

A data table is firstly generated, which lists all the HDL model ports ( $IN = \{in_1, \dots, in_n\}$ ,  $OUT = \{out_1, \dots, out_m\}$ ) as well as the corresponding data size and type. A C++ data structure is generated by mapping all the elements of the data table (with the exception of clock) into C++ structure fields (i.e.,  $el\_type_x in_1, \dots, el\_type_y in_n, el\_type_z out_1, \dots, el\_type_t out_m$ ), by exploiting the C++ data type extended library (see Section 7.1) for mapping HDL types into C++ types.

The C++ communication protocol is an *abstracted* version of the HDL model communication protocol, and may consist of one or more invocations of `main_IP()` to read inputs, elaborate, and return the results.

#### 4.1. Mapping of HDL computational phases to C++ computational cycles

The generation of the C++ communication protocol via abstraction relies on the concept of *computational phase* [Bombieri et al. 2007] and on the mapping of HDL computational phases into C++ computational cycles.

A computational phase of an RTL model is defined as a sequence of EFSM states that must be consistently traversed to get the input data (input sub-phase), elaborate them (elaboration sub-phase), and finally provide the related output result (output sub-phase). During the input sub-phase, the update functions of the traversed transitions read input data and control lines without performing any further elaboration. Then, data is manipulated in the elaboration sub-phase without reading new values from inputs neither writing on outputs. Finally, in the output sub-phase, the update functions do not modify the computation result anymore, while control and data output lines are written according to the communication protocol. The identification of the computational phases in a EFSM is automatic [Bombieri et al. 2007]. Figure 3 shows three different examples of computational phase and the corresponding input/elaboration/output sub-phases.

The C++ communication protocol is generated by mapping each computational phase identified in the HDL model into one or more computational cycles of the C++ model. During a C++ computational cycle, the C++ model can read one payload of input data (input sub-phase) and write one payload of output data (output sub-phase). The structure `io_struct` is used for exchanging data during the input and output sub-phases. At the end of each output sub-phase, a control flag (`flag_return`) is set to true to terminate the computational cycle and to allow the `main_IP()` to return (see Section 7.2).

Formally, given an input sub-phase in which the update function of the traversed transition ( $R$ ) reads data on input ports  $in_{1,\dots,i}$ , an elaboration sub-phase in which data is elaborated in the update function(s) ( $E$ ) and an output sub-phase in which the update function of the traversed transition ( $W$ ) writes the result on ports  $out_{1,\dots,j}$ , the C++ communication protocol is defined as the sequence:

$$(R\_in\{1,\dots,i\} \rightarrow E \rightarrow W\_out\{1,\dots,j\}) \quad (1)$$

which is performed by one C++ computational cycle (i.e., one invocation to `main_IP()`).

Figure 3 shows some examples. In Figure 3(a), the RTL model consists of a synchronous process, sensitive to the clock `clk`. At each clock cycle, the model reads input data (input sub-phase), elaborates them (elaboration sub-phase), and writes on the output port (output sub-phase). The HDL computational phase is one clock cycle long, and it is mapped into one C++ computational cycle. In the generated C++ model, the input data reading, elaboration and return of results are performed in one invocation of the `main_IP()`.

In Figure 3(b), the RTL mode still consists of a synchronous process, sensitive to the clock `clk`. In this case, the computational phase is  $2y + 3$  clock cycles long. That is, the model reads input data in the first clock cycle, it elaborates such data in a number of clock cycles ( $2y - 1$  cc), and writes the results in one clock cycle. Also in this case, the computational phase is mapped into one C++ computational cycle, that is, the input data reading, elaboration and return of results are performed in one invocation of the `main_IP()` in the generated C++ model.

In Figure 3(c), the RTL mode still consists of a synchronous process, sensitive to the clock `clk`. The input sub-phase is two clock cycles long, i.e., the model reads two data tokens from the same input port and, then, it starts the elaboration sub-phase. The model writes the resulting two data tokens in the same output port in two clock cycles. In this case, the input sub-phase must be mapped into two different C++ computational cycles, i.e., two invocations of `main_IP()` in the generated C++ model. The first

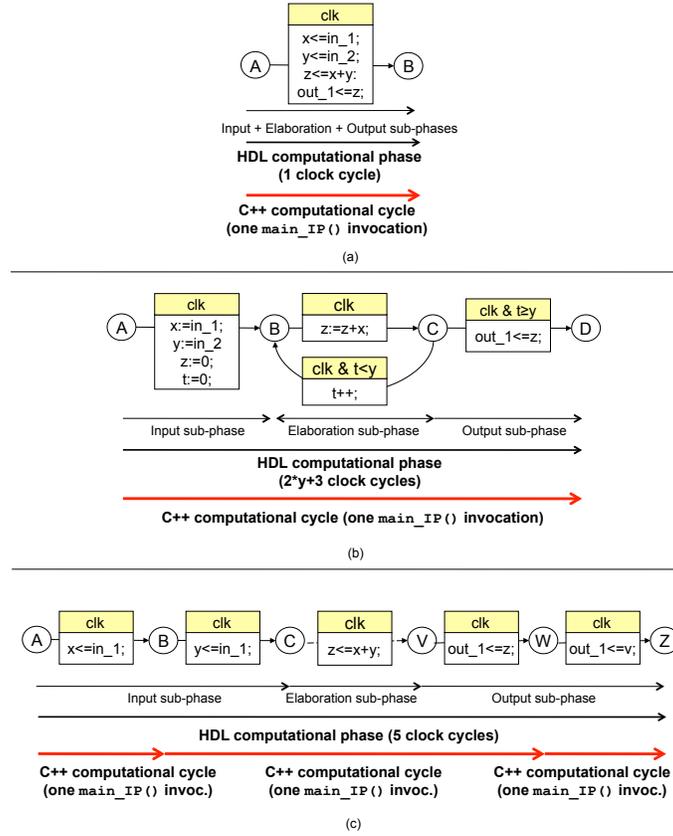


Fig. 3. HDL computational phases vs. C++ computational cycles

invocation is fired to read the first data token, while it does not perform any elaboration and return of results. The second invocation is fired to read the second data token, to carry on with the data elaboration, and to return the first data token. The second resulting data token needs on more invocation of `main_IP()` to be returned. As a result, an HDL computational phase is mapped into three C++ computational cycles. This situation exists when more input/output data tokens are read/written on the same port at RTL, and the RTL interface is preserved in the C++ model. To map the HDL computational phase into one C++ computational cycle would require a modification of the C++ interface. This modification may be not automatic, and it is not implemented in the proposed approach.

In general, one HDL computational phase must be mapped into more C++ computational cycles when an input sub-phase consists of more than one consecutive transitions to read data on the same input port before moving to the elaboration sub-phase:

$$(R1\_in\{i\} \rightarrow \dots \rightarrow Rn\_in\{i\} \rightarrow E \rightarrow W\_out\{j\}) \quad (2)$$

The same concept applies to the output sub-phase:

$$(R\_in\{i\} \rightarrow E \rightarrow W1\_out\{j\} \rightarrow \dots \rightarrow Wm\_out\{j\}) \quad (3)$$

Other similar situations are represented by HDL computational phases in which input and elaboration sub-phases alternate before moving to the output sub-phase (or elaboration and output sub-phases alternate after an input sub-phase), as follows:

```

C++ soft driver
// C++ computational cycle 1 (A -> B)
// set input values
io_struct.in_1=v1;
io_struct.in_2=v2;
//main_IP call
main_IP(io_struct);
// get results
vr=io_struct.out_1;

```

(a) – (b)

```

C++ soft driver
// C++ computational cycle 1 (A -> B)
// set input values
io_struct.in_1=v1;
//main_IP call 1
main_IP(io_struct);

// C++ computational cycle 2 (B -> W)
// set input values
io_struct.in_1=v2;
//main_IP call 2
main_IP(io_struct);
// get results
vr1=io_struct.out_1;

// C++ computational cycle 3 (W -> Z)
//main_IP call 3
main_IP(io_struct);
// get results
vr2=io_struct.out_1;

```

(c)

Fig. 4. C++ soft drivers of the examples of Fig. 3

$$(R1\_in\{i\} \rightarrow E_1 \rightarrow \dots \rightarrow Rn\_in\{j\} \rightarrow E_n \rightarrow W\_out\{t\}) \quad (4)$$

$$(R\_in\{i\} \rightarrow E_1 \rightarrow W1\_out\{t\} \rightarrow \dots \rightarrow E_m \rightarrow Wm\_out\{u\}) \quad (5)$$

#### 4.2. Generation of the communication protocol

The interface handshaking of the original HDL RTL model may require more than one HDL computational phase to read input data, to elaborate, and to write the final results. Each HDL computational phase may be mapped into one or more C++ computational cycles (see Section 4.1). As a result, the generated C++ application may require more than one invocation to the `main_IP()` to read data tokens, to elaborate, and to return the results. Such a sequence of invocations that implements the handshaking for the C++ model is an additional SW layer, which we call *soft driver*.

The automatic identification of HDL computational phases [Bombieri et al. 2007] and the mapping of HDL computational phases into C++ computational cycles allows us to generate the soft driver. The main idea of the generation algorithm is the following. Starting from the EFSM representation of the RTL model, the algorithm visits the EFSM graph. For each identified HDL computational phase, the algorithm visits the sequence of EFSM states representing such a phase. Depending on the recognized sequence (which can be one of the five defined in Section 4.1), the algorithm enriches the driver code by adding:

- One `main_IP()` invocation for each sequence of class (1). The input data ( $in\{1, \dots, i\}$ ) is exchanged during the invocation, which includes the elaboration and terminates by returning the results ( $out\{1, \dots, j\}$ );

- A number  $n$  of `main_IP()` invocations equals to the number of input sub-phases ( $R\_in\{i\}$ ) for each sequence of class (2). The input data tokens are exchanged through  $n$  invocations. The last invocation includes the elaboration and output sub-phases;
- A number  $m$  of `main_IP()` invocations equals to the number of output sub-phases ( $W\_out\{j\}$ ) for each sequence of class (3), similarly to (2);
- A number  $n$  of `main_IP()` invocations equals to the number  $n$  of partial computational phases (input + elaboration sub-phases ( $R\_in\{i\} - > E$ )) for each sequence of class (4). The input data tokens are exchanged and elaborated through  $n$  invocations. The last invocation includes the output sub-phases;
- A number  $m$  of `main_IP()` invocations equals to the number  $m$  of partial computational phases (elaboration + output sub-phases ( $E - > W\_out\{j\}$ )) for each sequence of class (5), similarly to (4).

Figure 4 shows the soft drivers of the examples of Figure 3. The soft drivers of the examples (a) and (b) are equal as both are related to an HDL communication protocol with one HDL computational phase. An external caller exploits the driver to set the input values, to call the `main_IP()`, and to get back the results. Figure 3.c represents a soft driver related to an HDL communication protocol with three computational phases. In this case the external caller invokes the `main_IP()` to set the first input value ( $v1$ ) and to move the computation from state  $A$  to  $B$ . Then, since the second input value ( $v2$ ) must be passed through the same input port (`in_1`) the caller must set again the `io_struct` and call the `main_IP()`, thus moving the computation from  $B$  to  $W$ . Since  $W$  is an output state, the caller can read the result values on the `io_struct`. Finally, the computation phase ends with a third call to `main_IP()` ( $W$  to  $Z$ ) to get other results since  $Z$  is an output state. The third call is due to the fact that the second result value is given back by the module on the same output port (`out_1`).

## 5. EFSM GENERATION

An EFSM [Cheng and Krishnakumar 1996] is a transition system that allows a more compact representation of the design states with respect to the more traditional finite state machine (FSM). The EFSM model is widely used for modeling complex systems like reactive systems [Koo et al. 1999], communication protocols [Katagiri et al. 2000], buses [Zitouni et al. 2006] and controllers driving data-path [Guerrouat and Richter 2006].

**DEFINITION 1.** *An EFSM is defined as a 5-tuple  $M = \langle S, I, O, D, T \rangle$  where:  $S$  is a set of states,  $I$  is a set of input symbols,  $O$  is a set of output symbols,  $D$  is a  $n$ -dimensional linear space  $D_1 \times \dots \times D_n$ ,  $T$  is a transition relation such that  $T : S \times D \times I \rightarrow S \times D \times O$ . A generic point in  $D$  is described by a  $n$ -tuple  $x = (x_1, \dots, x_n)$ ; it models the values of the registers internal to the design.*

A pair  $\langle s, x \rangle \in S \times D$  is called *configuration* of  $M$ , while an operation on an EFSM  $M = \langle S, I, O, D, T \rangle$  is defined as follows:

**DEFINITION 2.** *If  $M$  is in a configuration  $\langle s, x \rangle$  and it receives an input  $i \in I$ , it moves to the configuration  $\langle t, y \rangle$  iff  $((s, x, i), (t, y, o)) \in T$  for  $o \in O$ .*

In an EFSM, each transition is associated with a couple of functions (i.e., an *enabling function* and an *update function*) acting on input, output and register data. The enabling function expresses a set of conditions on data, while the update functions consist of a set of statements performing operations on data.

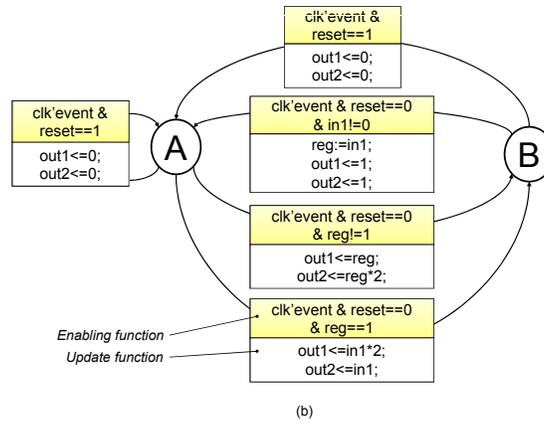
**DEFINITION 3.** *Given an EFSM  $M = \langle S, I, O, D, T \rangle$ ,  $s \in S, t \in T, i \in I, o \in O$  and the sets  $X = \{x | ((s, x, i), (t, y, o)) \in T \text{ for } y \in D\}$  and  $Y = \{y | ((s, x, i), (t, y, o)) \in T \text{ for } x \in X\}$ , the enabling and update functions are defined respectively as:*

```

if (clk'event) then
  case STATE is
    when A =>
      if (reset==1) then
        out1<=0;
        out2<=0;
        NEXT_STATE <= A;
      else if (reset==0 and reg!=1) then
        out1<=reg;
        out2<=reg*2;
        NEXT_STATE <= B;
      else if (reset==0 and reg==1) then
        out1<=in1*2;
        out2<=in1;
        NEXT_STATE <= B;
      endif;
    when B =>
      if (reset==1) then
        out1<=0;
        out2<=0;
        NEXT_STATE <= A;
      else if (reset==0 and in1!=0) then
        reg:=in1;
        out1<=1;
        out2<=1;
        NEXT_STATE <= A;
      endif;
    when others =>
      out1<=0;
      out2<=0;
      NEXT_STATE=STATE;
    end case;
  end case;
endif;
.VHDL

```

(a)



(b)

Fig. 5. Example of VHDL model (a) and the corresponding EFSM representation (b)

$$e(x, i) = \begin{cases} 1 & \text{if } x \in X; \\ 0 & \text{otherwise.} \end{cases}$$

$$u(x, i) = \begin{cases} (y, o) & \text{if } e(x, i) = 1 \text{ and} \\ & ((s, x, i), (t, y, o)) \in T; \\ \text{undef.} & \text{otherwise.} \end{cases}$$

Figure 5 gives an example of VHDL code and its representation through EFSM. In the state transition graph of Figure 5(b), a transition is fired only if all conditions in the enabling function are satisfied, bringing the machine from the current state to the destination state and performing the operations included in the update function.

The EFSM representation of the RTL IP model can be automatically extracted from the corresponding RTL description, as described in [Cheng and Krishnakumar 1996]. The methodology generates a single EFSM for a complete RTL model.

Figure 2.2 shows the representation of the IP model by means of EFSMs, which includes the interface  $I$ , the set of synchronous processes  $PS = \{ps_1, \dots, ps_i\}$  and the set of asynchronous processes  $PA = \{pa_1, \dots, pa_j\}$ . The sequential statements have not

been reported in Figure 2 for the sake of clarity, but they have to be seen as update functions of the process transitions, as described in Section 6.

Finally, each concurrent statement  $cs \in CS$  containing an expression  $e$  (e.g.,  $a \Leftarrow b$  AND  $c$ ) is converted into an equivalent asynchronous process  $p$  that is sensitive to the right-hand signals of the expression (e.g.,  $b$ ,  $c$ ) and that has the expression  $e$  as the unique sequential statement. For example, the following set of concurrent statements:

```
B: BLOCK BEGIN --concurrent area
  a <= b AND c; --cs1
  d <= a AND c; --cs2
END BLOCK b;
```

is converted into the equivalent set of asynchronous processes:

```
p_1: PROCESS (b, c) BEGIN
  a <= b AND c; --ss1
END PROCESS p_1;

p_2: PROCESS (a, c) BEGIN
  d <= a AND c; --ss2
END PROCESS p_2;
```

The model implemented by block  $B$  is equivalent to the model implemented by the two asynchronous processes  $p_1$  and  $p_2$ , as proved in [Mentor Graphics 1994]. Without loss of generality, we refer to  $PA = \{pa_1, \dots, pa_j\}$  as the set composed of the original asynchronous processes and the asynchronous processes generated from concurrent statements.

General events ( $e_i$ ) are the guards for the transitions of the asynchronous processes  $pa_1, \dots, pa_j$ , while a clock edge is the guard for all the transitions of the synchronous processes  $ps_1, \dots, ps_i$ . In both cases, each fired transition causes a set of sequential statements  $ss_i \in SS$  to be executed.

## 6. MERGE OF PROCESSES

This step of the methodology aims at optimizing the model structure, by reducing the number of processes. Two HDL processes that are sensitive to the same set of events behave as concurrent statements, and the two sets of sequential statements contained in each process bodies are evaluated as if they occurred simultaneously<sup>1</sup>. Finally, they always resume simultaneously in the same instant during simulation [VHDL 1994; Verilog 2006; SystemC 2006].

The merging algorithm reduces the number of processes by following the principle of concurrent statements run on a single processor system. With just one processor, concurrent processes are not actually evaluated in parallel on the hardware simulator. HDLs use the concept of *delta delay* to keep track of processes that should occur in a given timestamp but that are actually evaluated in different machine cycles. Concurrent processes are sequentially executed *non-deterministically* and the order of execution does not affect the result [Mentor Graphics 1994; OSCI 2002].

Often, concurrent systems may have locally non-deterministic properties that are intentional, still exhibiting deterministic global behaviour. As an example, at a very low level, individual packets transmitted over a network may take unpredictable routes to reach their destination or may be lost altogether, while, globally, reliable connections are established over the network.

<sup>1</sup>In general, this is true if no `wait` statement appears within the process bodies. However, the EFSM generation process guarantees this assumption, as it translates each process containing `wait` statements into an equivalent process with no `wait`, by adding more EFSM states.

On the other hand, it is possible that local non-deterministic behaviour may result in undesirable global non-deterministic behaviour. An example is given by two HDL processes that are scheduled to run at the same time and that simultaneously attempt to update a global variable. In this case (i.e. a non-properly designed system) the variable value is unpredictable.

In general, non-determinism may exist in any HDL model. In some cases this is intentional since it represents a property of the system, while, in other cases, it is undesirable since it represents a design flaw.

In this work, we assume that non-determinism is introduced in the RTL IP descriptions since the execution order of processes within a particular simulation phase (or part of a simulation delta cycle) is unspecified (or is "non-deterministic"). In a properly designed system, this aspect of the HDL does not affect the overall system behaviour.

The proposed merging algorithm exploits this concept to join all these processes into macro-processes. The blocks of sequential statements composing the body of each process are sequentially appended to a single body. The algorithm statically resolves the non-determinism by choosing one of the possible orders of execution, without affecting the execution correctness. For example, the following two concurrent processes:

```
p_1: PROCESS (a, b) BEGIN
  c <= a AND b; --ss1_1
  x := y + z;   --ss1_2
END PROCESS p_i'ui1;
```

```
p_2: PROCESS (a, b) BEGIN
  d <= a OR b;  --ss2_1
  x := y + z;   --ss2_2
END PROCESS p_2;
```

are merged into the macro-process:

```
p_1_2: PROCESS (a, b) BEGIN
  c <= a AND b;      --ss1_1
  x_1 := y_1 + z_1; --ss1_2
  d <= a OR b;      --ss2_1
  x_2 := y_2 + z_2; --ss2_2
END PROCESS p_1_2;
```

where the merging algorithm solves naming, scope and visibility in case there is any component (such as variables  $x$ ,  $y$ , and  $z$  in the example above) with the same name into both processes. The algorithm iteratively merges processes until there are not two processes with the same sensitivity list. As result, the initial set of processes (i.e.,  $PS \cup PA$ ) is reduced to  $\{ps, pa_1, \dots, pa_{j-x}\}$  (see Figure 2.3), where  $ps$  is the synchronous macro-process in which all the clocked sequential statements have been merged, while  $pa_1, \dots, pa_{j-x}$  is the minimum set of asynchronous processes.

## 7. MAPPING OF HDL STATEMENTS INTO C++ STATEMENTS

Each HDL statement that composes the EFSM model of the IP is translated into a corresponding C++ statement (see Figure 2.4). The translation algorithm relies on the following set of rules, which define a mapping between HDL and C++ statements by considering the syntactic and semantic differences between the two languages:

- (1) Each HDL signal  $s_i$  is translated into a two variable structure  $sig_i = \{sig.old, sig.new\}$ . The two variables represent the old and the current values of the signal in the C++ model simulation.
- (2) Each set of HDL sequential statements  $SS_i$  is translated into a set of C++ statements. The translation of sequential statements is merely syntactic.

- (3) The synchronous macro-process  $ps$  is translated into the C++ function  $f_{ps}()$ . The function body contains the set of C++ statements translated from the sequential statements of  $ps$ .
- (4) Each asynchronous process  $pa_i$  is translated into a function  $f_{pa_i}()$ . The function body contains the set of C++ statements translated from the sequential statements of  $pa_i$ . Each set of signals to which the process  $pa_i$  is sensitive (e.g., signal  $s$ ) is translated into a set of event flags (e.g., *bool event\_s*). Such flags give information on the combinational path represented by any process  $p$  (that writes on  $s$ ) and  $pa_i$ . The flags will be exploited to execute the corresponding functions in the correct sequential order during the C++ model simulation.

Besides statements mapping, the methodology aims at generating a structured C++ code that (i) relies on efficient and more abstract data types than the HDL ones, and (ii) implements an internal scheduling kernel to make the C++ code independent from any operating system. A bit-accurate data type library is defined to implement, in C++, all the HDL data types and the corresponding operators that are not native of C++ (e.g., *logic\_vector*), as explained in Section 7.1. Two additional functions (i.e., *scheduler()* and *update\_event\_queue()*) are defined into the C++ model for implementing an abstract version of the HDL scheduling algorithm, as explained in Section 7.2.

### 7.1. The C++ bit-accurate data type library

HDLs support hardware modeling with a large number of data types, which recreate low level behaviors of the target physical circuit. SystemC is the de-facto reference standard that implements such a bit-accurate data types in C++. Nevertheless, it has been proven that the SystemC data type implementation does not guarantee the best simulation performance [Ecker et al. 2007].

We propose a new C++ bit-accurate data type library that provides highly optimized HDL types and that has been implemented by exploiting advanced optimization techniques, like compile-time optimizations based on C++ templates. The methodology supports two-valued semantics when the given RTL IP is synthesizable and correct (i.e., the generated C++ code preserves the two-valued semantics). On the other hand, when the RTL IP description is not synthesizable (i.e., the code is enriched with debugging and verification statements, which make use of meta values like *Xs* and *Zs*), the methodology implements a code introspection algorithm to get rid of such statements. If the introspection step is successful, the semantics of the generated C++ code is two-valued. If the algorithm fails, the semantics is multi-valued, which slows down the performance but guarantees the code generation.

The library consists of two data types: a 4-valued logic vector class and a 2-valued bit vector class. Both classes are templated, taking one integer parameter which indicates the bitwidth (i.e., the number of elements belonging to the vector).

In order to achieve a significant performance improvement, the following solutions have been adopted when implementing these data types:

- (1) *No heap memory allocation*. A statically allocated array of unsigned integers is employed as underlying data structure used to store vector elements. The size of such an array can be computed at compile time, since it depends only on the width of the template argument. For example, a bit vector having width  $W$  contains the following declarations:

```
// static constant that stores the number of
// chunks required to accommodate W bits:
static const unsigned int CHUNKS_NUMBER =
    = W / (sizeof(chunk_t) * 8) +
```

```

+ (W % (sizeof(chunk_t) * 8) ? 1 : 0);
... unsigned int _chunks[CHUNKS_NUMBER];

```

This solution allows the heap memory allocation to be avoided, since it causes additional overheads at runtime and prevents compiler optimizations, thus resulting in a performance hit.

- (2) *Operations performed on words instead of single bits.* The choice of unsigned integers as data structure allows data type operators to be implemented over words as a whole, thus avoiding iterative operations on each single vector element. For example, the bitwise negation for bit vectors is implemented as follows:

```

for (register unsigned int i = 0;
     i < CHUNKS_NUMBER; ++i)
    result._chunks[i] = ~(_chunks[i]);

```

This is achieved by carefully implementing operations on architecture-dependent words by properly using bitwise operations and shifts. These are among the fastest instructions to be executed on any machine, since they take advantage of word-sized registers and optimized ALUs to be executed in a single CPU operation.

- (3) *Mapping of a logic value on two separate bits.* Logic vectors have been implemented by using two separate arrays of unsigned integers. Each logic value is associated with two bits, one per array, to represent the four possible values. For example, the logic vector class consists of the following variables:

```

unsigned int _lower_chunks [CHUNKS_NUMBER];
unsigned int _upper_chunks [CHUNKS_NUMBER];

```

In this way, operations on logic values are implemented in terms of bitwise and shift operations on architecture-dependent words. Instead of implementing operations through lookup tables, which store the pre-computed result for every pair of operands, we implemented operators by relying on bitwise operations (such as AND, OR and NOT) carried out on integers. The proper sequence of bitwise operations has been obtained by using Karnaugh maps on the operations to be performed, thus replacing the lookup tables. This replacement allows us to replace a memory access (which usually spans through multiple CPU cycles) by a short sequence of bitwise operations on words of the CPU architecture, which can be executed in a single CPU cycle.

For example, the implementation of the bitwise negation operator can be sketched as follows:

```

for (register unsigned int i = 0;
     i < CHUNKS_NUMBER; ++i) {
    result._lower_chunks[i] = _upper_chunks[i] &
    & (~(_lower_chunks[i]));
    result._upper_chunks[i] = _upper_chunks[i];
}

```

- (4) *Minimal class hierarchy.* In order to reduce the impact of managing parent constructors and destructors at runtime, the class hierarchy has been kept to the bare minimum.

VHDL	C++
bit, boolean, <i>std_logic</i>	bool
bit_vector<N>, <i>std_logic_vector&lt;N&gt;</i>	bool[N]
char	char
integer	int/ long int/ long long int
unsigned	unsigned int/ long int/ long long int
<i>real</i>	float/double/long double

(a)

Verilog	C++
reg, uwire, <i>net{wire, wand, wor, etc.}</i>	bool
reg<N>, uwire<N>, <i>net{wire, wand, wor, etc.}&lt;N&gt;</i>	bool[N]
integer	int/ long int/ long long int
<i>real</i>	float/double/long double

(b)

SystemC	C++
bool, sc_bit, <i>sc_logic</i>	bool
sc_bv<N>, <i>sc_lv&lt;N&gt;</i>	bool[N]
char	char
sc_int<N>	int/ long int/ long long int
sc_uint<N>	unsigned int/ long int/ long long int
<i>sc_fixed&lt;wl, iwl, q, o, n&gt;</i>	float/double/long double

(c)

Fig. 6. Mapping of HDL data types into C++ data types ((a) VHDL, (b) Verilog, (c) SystemC) data types

Figure 6 shows the mapping of the most important HDL data types into the proposed C++ bit-accurate data type library. For VHDL, the figure reports the native VHDL types as well as the data types implemented into the IEEE 1164 library (e.g., `std_logic` and `std_logic_vector<N>`). SystemC presents a large set of HDL data types besides the native ones (some of them omitted for the sake of clarity), while Verilog relies on a more reduced set of data types.

Mapping HDL two-valued logic data types into C++ native data types is straightforward. For example, `bit`, `boolean` or `bit_vector` of VHDL, `reg`, `reg<N>` of Verilog, and, `sc_bit`, `sc_bv` of SystemC are mapped into the native two-valued logic types of C++ (i.e., `bool` or to C++ vectors of `bool`).

Integer data types are mapped into native integer data types according to their size. All functions and operators are preserved. A new class (`long_integer_class`) has been defined to implement integer with size bigger than 64 bits in terms of instances of native integer types.

The HDL multi-valued logic types as well as the HDL fixed point types (in red italics in Figure 6) are abstracted into more efficient C++ data types, as explained in the next subsection.

*7.1.1. The data type abstraction.* The data type abstraction aims at abstracting the multi-valued logic data types into the two-valued logic of the C++ native data types and the fixed point types into floating point types.

The multi-valued logic extends the two-valued logic (i.e., '0', '1') with a set of *meta values* (i.e., 'U', 'X', 'Z', 'W', '-', 'L', 'H' in VHDL, and, 'X', 'Z' in Verilog

and SystemC). Meta values are used mainly for debugging purposes and for simulating hardware-specific behaviors, such as, uninitialized, unknown, high impedance values, and so on.

'X' and 'Z' are the most commonly used meta values and the key concepts of their abstraction is explained in the following. Type abstraction of the other values is part of our current and future work<sup>2</sup>.

The *unknown value* 'X' is used to express that a value of a logic type is uncertain. It is not explicitly used for implementing the RTL model behavior since it does not map any actual circuit value. Rather, it is used for low-level debugging. If a 'X' value is observed after the design initialization (e.g., a reset phase) or during execution, it means that the circuit most likely contains a bug, since a non-deterministic behavior has been introduced. Therefore, an RTL model can have an explicit use of 'X' only in conditional statements introduced for verification. When applying the proposed type abstraction methodology, such debugging statements are removed, still obtaining a functionally equivalent design.

The *high-impedance value* 'Z' is used for tri-state signals (i.e., signals with more than one driver). When a driver writes a 'Z' on a signal, it allows other drivers to set the value. The explicit use of 'Z' in a RTL model can occur in two cases only: in conditional statements inserted for debugging or in write operations. Debugging statements are removed (as done for unknown values) by obtaining a functionally equivalent design. Write statements that assign 'Z' to a variable/signal can also be removed since we assume that, in a correct design, there exist at least one statement that assigns a value different from 'Z' to such a variable/signal.

By removing statements that make explicit use of 'X' and 'Z', the low level debugging features (e.g., concurrency debugging) of the IP code are removed. However, the proposed methodology aims at automatically generating C++ code implementing the high-level IP functionality rather than its low level debugging features.

The 'X' meta value can also be implicitly introduced during simulation by *resolution functions* [VHDL 1994; SystemC 2006]. A resolution function is a HDL-dependent function which handles the multi-valued logic. In the most common case, a 'X' can be generated when more than one driver tries to set different values, which are not 'Z'. The generation of 'X' notifies a design error as it represents a non-deterministic behavior of the actual circuit (e.g., the SystemC kernel by default stops the simulation whenever such a concurrent assignment happens). The concurrent behaviour and the corresponding resolution functions are abstracted so that the C++ model generates '0' or '1' non-deterministically instead of 'X' and it notifies a warning of possible design errors. It is important to note that the reported analysis holds supposing that the RTL model is synthesizable. In case of non-synthesizable RTL models, the C++ is generated without data type abstraction.

Fixed point data types (e.g., VHDL or Verilog `real` and SystemC `sc_fixed`) are generally used in signal processing algorithms to reduce the cost of hardware while increasing throughput. In the proposed approach, fixed point types are mapped into C++ floating-point types (`float`) with the aim of exploiting the floating point units of the CPU. Nevertheless, such a conversion involves an approximation that depends on both the C++ compiler and the CPU architecture. As a consequence, in specific cases, the fixed to floating point mapping may lead to functionally different results (e.g., `if (result == expectedResult)`). The mapping is conservatively applied to RTL

<sup>2</sup>The current version of the methodology abstracts the 'X' and 'Z' meta values only (i.e., it fully supports the Verilog and SystemC logic datatypes, while partially supports the VHDL logic datatypes). Currently, if any other meta value (of VHDL) is explicitly used in the RTL description, the whole logic type class is replicated into the C++ code with a partial impact on performance.

```

1: -- Initialization phase
2: Initialize();
3: -- Simulation phase
4: while (simulation_time < simulation_end_time) do
5:   -- HDL simulation cycle
6:    $T_c := T_n$ ;
7:   Update_signals();
8:   for each process  $p_i$  of the model to simulate do
9:     if  $p_i$  is currently sensitive to a signal  $s$  and an event  $e_s$  occurred on  $s$  in this simulation
       cycle then
10:      process_queue.enqueue( $p_i$ );
11:    end if
12:  end for
13:  for each  $p_i \in$  process_queue do
14:     $p_i$  executes until it suspends;
15:    process_queue.dequeue( $p_i$ );
16:  end for
17:   $T_n := \text{earliest\_of}\{\text{simulation\_end\_time}, \text{next\_event\_time}, \text{next\_process\_time}\}$ ;
18:  // If  $T_n = T_c$  then the next simulation cycle (if any) will be a delta cycle.
19: end while

```

Fig. 7. HDL scheduling algorithm

models that do not have floating point comparisons (or comparison results) in conditional statements. The whole HDL fixed point type class has been re-implemented through C++ `double` and `long double` to support all the other cases.

## 7.2. Abstraction of HDL scheduler

By applying the four translation rules presented at the beginning of Section 7, the abstraction algorithm automatically translates HDL statements (i.e., declaration, concurrent, and sequential) into equivalent C++ statements.

To preserve the RTL IP semantics, the C++ functions (`fps()`, `fpa1()`, ..., `fpaj()`) generated from the RTL processes (`ps`, `pa1`, ..., `paj`) have to be executed in the same *partial* order as the corresponding RTL processes are executed in the RTL model. Concurrent processes can execute in a non-deterministic order, while the order between non concurrent processes must be preserved.

Figure 7 summarizes the main steps of a HDL scheduling algorithm. The algorithm may differ in more than one detail according to the semantics of the specific HDL language (e.g., immediate notification handling in SystemC, signal assignment handling in VHDL, etc.). Since such differences do not alter the proposed methodology, without loss of generality, this article refers to such an algorithm for explaining the scheduling abstraction.

The execution of a HDL model consists of an *initialization phase* (which sets the initial value of each signal and executes each process until it suspends) followed by the *simulation phase*, which iteratively executes processes until the end of the simulation (line 4 of Figure 7). Each such iteration is called *simulation cycle*. In each cycle, the current time,  $T_c$ , is firstly set equal to the time of the next simulation cycle,  $T_n$  (line 6). Then, the values of all signals of the model are updated (line 7). If, as a result of this computation, an event occurs on a given signal, all processes sensitive to that signal are added to the *runnable queue* (*update step*). Then, all runnable processes resume and execute as part of the simulation cycle (lines 8 - 16) (*evaluate step*). Finally, during the *time update phase*, time of the next simulation cycle  $T_n$  is determined (line 17) by setting it to the earliest of (i) the time at which simulation ends, (ii) the next time at which an event occurs, or (iii) the next time at which a process resumes. If  $T_n = T_c$ , the

```

1: // C++ computational cycle
2: while (flag_return = false) do
3:   // C++ simulation cycle
4:   fps() executes;
5:   while (function_queue is not empty) do
6:     update_event_queue();
7:     for each function fpai do
8:       if at least one event of the corresponding set of event = true then
9:         function_queue.enqueue(fpai);
10:      end if
11:     end for
12:     for each fpai ∈ function_queue do
13:       fpai executes;
14:       function_queue.dequeue(fpai);
15:     end for
16:   end while
17: end while

```

Fig. 8. C++ scheduling algorithm

next simulation cycle (if any) will be a *delta cycle*. The simulation stops when there are no more timed notifications.

The C++ scheduler is implemented by abstracting away timing details (i.e., timed events and clock) while preserving the dynamic scheduling semantics. That is, the process (functions) execution order is known at run time and processes (functions) are woke up when there has been an event to which they are sensitive.

Figure 8 summarizes the main steps of the C++ scheduling algorithm, which is implemented as extra statements in the generated C++ model. The C++ scheduler execution consists of a *C++ computational cycle*, which runs one or many *C++ simulation cycles*. The C++ computational cycle starts when the *main\_IP()* function of the C++ model is invoked, and returns when the *flag\_return* is set to true (line 2).

A C++ simulation cycle starts by executing function *fps()* (line 4) on new input values. Then, iteratively, an abstracted version of the event updating is executed. For each signal  $s_i$  (i.e., internal signal or input port) if  $s_i.old \neq s_i.new$  then the corresponding event flag is set to true (see rule 4)(line 8). As a result, all functions  $fpa_i()$  with the corresponding event flag set to true are put into the *function\_queue* (lines 7-11) and are sequentially executed (lines 12-15). The queues are bounded and it is always possible to bound their size at compile time.

Consider, for example, the IP block in Figure 9. Figure 9(a) reports a simple example of RTL IP in VHDL. Figure 9(b) represents the RTL IP model of such a block through a graph, each process being a vertex and each signal being an oriented edge. The graph represents the synchronization and communication net among processes. The RTL IP block consists of four synchronous processes (ps0-ps3), three asynchronous processes (pa0-pa2), two input ports (in1, in2), two output ports (out1, out2), and seven internal signals (sig1- sig7). Figure 9(c) represents the corresponding process execution order, by underlining update and evaluate steps, simulation cycles, and delta cycles. Figure 9(d) shows an overview of the generated C++ code, which implements the IP functionality in terms of functions ( $fps()$ ,  $fpa_1() - fpa_3()$ ) and scheduling.

For many target applications of the proposed methodology, timing constraints may be on the worst-case rather than on the average-case performance of the software. As an example, IPs with communication protocol that has strict timing requirements like VGA controllers, filters, etc.. The generated code may have variability of performance, even though quite limited w.r.t. the IP latency. Such a variability depends on the RTL model structure (number of processes, process synchronization and communication,

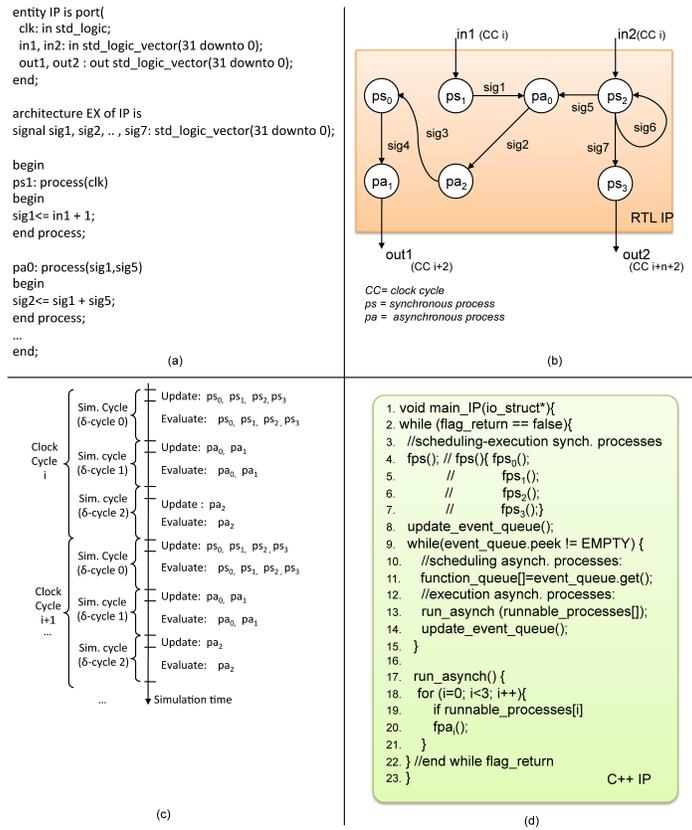


Fig. 9. (a) A simple example of RTL IP, (b) the process synchronization and communication graph of the RTL IP, (c) the process execution order, (d) the dynamic scheduling of functions in the C++ code.

etc). In any case, for how the process scheduling is implemented, it is always possible to measure the worst-case performance. In addition, many optimizations may be applied to improve the code performance. Nevertheless, the proposed approach aims more at maintaining readability (as much as possible) and portability of the code.

A computational cycle of the C++ model (one invocation of `main_IP()`) may perform the functional activity of the RTL IP over one or more simulation cycles. In particular, in case of one-to-one mapping (i.e., one C++ computational cycle for each HDL simulation cycle), the C++ is said to be cycle accurate. A computational cycle may also perform a number of clock cycles of functional activity of the RTL IP. When such a number is equal to the latency of the RTL IP, the cycle-accurate behaviour of the RTL model is said to be completely abstracted in the C++ model. The generation of the C++ communication protocol relies on such a mapping, as explained in the next section.

## 8. EXPERIMENTAL RESULTS

The proposed methodology has been implemented in *H2C++*, a tool built on the top of HIFSuite [EDALAB 2014], which parses RTL IPs implemented in VHDL or Verilog and generates C++ code<sup>3</sup>.

<sup>3</sup>*H2C++* does not parse SystemC models yet. Nevertheless, the proposed methodology can be applied to SystemC RTL descriptions.

Table I. Characteristics of the RTL IPs

CL	RTL IP block	PIs (#)	POs (#)	Gates (#)	FF (#)	HDL (loc)	RTL EFSM		Latency (#cc)	Th.put (data/cc)	Processes (#)		HDL source
							st(#)	tr(#)			Syn.	Asyn.	
1	LP_IIR_Filter	31	16	2,990	802	972	9	10	8	0.125	98	38	VHDL
	BM_Lambda	159	89	811	238	335	508	532	502	0.002	1	2	Verilog
	Lambda_roots	99	100	1,066	198	329	794	802	790	0.001	0	5	Verilog
	Omega_phy	228	193	8,735	1,322	1,595	302	308	294	0.003	17	4	Verilog
	Out_stage	11	20	498	177	274	10	10	8	0.125	2	0	Verilog
	Error_corr	301	83	7,736	1,855	1,666	156	172	130	0.008	6	11	Verilog
	DIV_FRS	35	33	248	19	58	15	21	2	0.5	4	4	VHDL
2	DIST_FILFRS	34	66	400	35	84	7	8	1	1	3	5	VHDL
	JPEG	20	27	92,056	1,435	7,103	118	136	92	1	241	826	Verilog
	FFT	92	114	87,397	1,359	3,335	54	212	3	1	16	11	VHDL
	DSPI	25	21	1,335	132	1,171	8	10	16	0.062	8	18	VHDL
	ADPCM	66	35	24,412	364	305	8	15	2	0.5	1	0	VHDL
	CRC	56	34	9,213	385	492	6	7	16	1	3	6	VHDL
	Root_FRS	35	33	682	59	119	6	7	16	0.063	3	3	VHDL
	GCD	67	65	636	51	100	3	4	1	1	1	21	VHDL
	ECC	25	32	993	79	175	5	6	1	1	4	0	VHDL

*H2C++* has been applied to a set of RTL IPs provided by industrial partners in the context of the TouchMore European Project [TouchMore 2013]. The RTL IPs have been grouped into two classes:

- (1) Custom RTL IPs. It includes IP blocks originally developed in VHDL/Verilog, and for which a C/C++ implementation was not available. They mainly implement filters and controllers. For all these blocks, we manually implemented a C++ version from scratch. We compared the performance of such a C++ code versus the C++ code automatically generated by *H2C++*.
- (2) Standard RTL IPs. This class includes IP blocks originally developed in VHDL/Verilog for which there was also a C/C++ implementation. For these blocks, we compared the existing C/C++ code versus the C++ code generated by *H2C++*.

Table I reports the structural characteristics of each RTL IP in terms of primary inputs and outputs (*PIs*, *POs*), gates, flip flops (*FF*), and lines of HDL code (*HDL*). Column *RTL EFSM* reports the number of states and transitions of the EFSMs extracted from the RTL IPs (see Section 5). Latency and throughput are reported in terms of clock cycles and output data per clock cycle, respectively. Columns *Synchronous* and *Asynchronous processes* report the number of processes composing the HDL model (asynchronous processes include concurrent assignments). All the analysed examples are single clock models. Support for multi-clock models is part of our future work.

Table II reports the characteristics of the C/C++ code, both manually and automatically generated by *H2C++*, in terms of C++ code lines and simulation time. Experiments have been conducted on a AMD A8 3870 with a Linux Ubuntu 11.04 Operating System. The C++ code was compiled with GCC 4.4.5 (-O3).

For the C++ code generated by *H2C++*, Table II also reports the number of C++ computational cycles (see Section 4.1) and the C++ code lines to implement the SW driver (if needed). Column *Data type abstraction* reports the HDL data types used in the RTL IP implementation and that have been mapped or abstracted into data types of the C++ bit-accurate library presented in Section 7.1.

All Bit and Integer data types of the VHDL blocks have been mapped into the native C++ types of the proposed library. The multi-valued VHDL standard logic and standard logic vector types and Verilog types have been abstracted into the two-valued C++ native types (see Section 7.1.1). We found the explicit use of the 'X' and 'Z' meta-values in some components of the *LP\_IIR\_Filter* and *FFT* benchmarks. They are used in reset states and in condition statements for debugging purpose. They have been abstracted, as explained in Section 7.1.1 without perturbing the IP functionality.

Table II. Characteristics and performance of the C/C++ code, both manual and generated by *H2C++*

CL	C++ IP block	Manual C/C++ code		H2C++ code					Simul. overhead (%)
		C++ (loc)	Simul. time (ms)	C++ code (loc)	Comp. cycles (#)	SW driver (loc)	Data type abstraction	Simul. time (ms)	
1	LP_IIR_Filter	1,337	1,162	1,442	1	-	Bit, Int, Std_logic/lv	1,341	15.40
	BM_Lambda	277	733	312	2	31	Logic/Logic vect.(4values)	772	5.32
	Lambda_roots	202	231	291	1	-	Logic/Logic vect.(4values)	239	3.46
	Omega_phy	1,122	2,695	1,343	3	48	Logic/Logic vect.(4values)	2,930	8.72
	Out_stage	211	181	237	1	-	Logic/Logic vect.(4values)	185	2.21
	Error_corr	989		1,242	3	39	Logic/Logic vect.(4values)	1,039	5.06
	DIV_FRS	22	1,450	42	1	-	Int, Std_logic/lv	1,551	6.97
2	DIST_Fil_FRS	37	1,460	72	1	-	Int, Std_logic/lv	1,635	11.98
	JPEG	1,142	1,952	40,295	1	-	Logic/Logic vect.(4values)	2,203	12.85
	FFT	876	410	3,246	4	42	Bit, Int, Std_logic/lv, Real	1,280	212.20
	DSPI	353	2,130	980	3	32	Bit, Int, Std_logic/lv	3,870	81.69
	ADPCM	271	3,960	262	2	18	Bit, Int	4,330	9.34
	CRC	235	3,490	420	4	46	Bit, Int, Std_logic/lv	5,360	53.58
	Root_FRS	18	260	98	1	-	Bit, Int, Std_logic/lv	370	42.31
	GCD	25	1,580	80	1	-	Bit, Int	1,770	12.03
	ECC	224	310	169	4	38	Bit, Int, Std_logic/lv	340	9.67

The functional verification of the generated C++ IPs versus the original RTL IPs has been performed via simulation through an automatic test pattern generator.

The *Real* data type was used in the RTL *FFT* benchmark. In particular, it was used for data comparisons in conditional statements and, thus, it has been mapped into the re-implemented class *real* rather than into *floating point* types.

Table II reports the comparison of performance between the C/C++ code implemented by hand and the C++ code generated by *H2C++*. The comparison is expressed in terms of simulation overhead introduced by running the automatically generated code with respect to the manual one.

In general, and as expected, the results show that the code implemented by hand is more efficient than the *H2C++* code. The difference of performance is more evident for the code implementing standard algorithms (class 2), for which the several and optimized implementations are available both in the industry IP libraries and third-party vendors. The difference is mainly due to the fact that the functionality of such IPs have been implemented at high-level of abstraction from scratch, by adopting native C/C++ data types, not bit-accurate operations, by disregarding interface and protocol constraints (size of I/O data interface), and with no temporal constraints for statement executions (e.g., critical paths handling, etc.).

The difference of performance between the manually and automatically generated codes is less evident for the RTL blocks implementing customized IPs (class 1). This is due to the fact that, even though the code has been implemented at high-level of abstraction from scratch, there were not many optimization alternatives for implementing the IP functionality. Starting from scratch to implement the IP block, the main differences between a manual C/C++ and a RTL code that may sensibly affect performance are related to (i) the partitioning of the IP functionality into HDL processes and (ii) the used data types and corresponding operators. In both cases, the merge of processes (Section 6) and the data type abstraction (Section 7.1) were fully applicable to all the benchmarks of class 1, thus generating a C++ code structurally similar to the code implemented by hand. In these cases, the simulation overhead is mainly due to the scheduling of the asynchronous processes, which consists of memory access statements to check for new events (see Section 7.2). We found that the overhead increases proportionally to the number of asynchronous processes (see for instance *LP\_IIR\_Filter* versus *Lambda\_roots* or *Out\_stage*).

For smaller RTL IPs (i.e., with few HDL processes), we found that the complexity of interface and communication protocol (e.g., handshaking phases longer than a clock

cycle) are the main causes of the simulation overhead (see for instance *BM\_Lambda*). This is due to the fact that a RTL computational phase is implemented by more C++ computational cycles, thus slowing down the I/O data exchange.

In general, we experimented that the efficiency of the automatic abstraction methodology and, thus, the performance of the generated code are not related to the size of the RTL model. Rather, they are related to the RTL IP structure (i.e., number and types of processes), interface, and communication protocol (number of computational phases).

For the RTL IPs of class 1, we obtained a simulation overhead ranging from a minimum of 2.21% of the *Out\_stage* block to a maximum of 11.98% of the *DIST\_Fil\_FRS*. We consider such an overhead negligible, since the code has been generated automatically.

The methodology has been applied also to standard RTL IPs (class 2) to understand how much the simulation overhead may increase in the worst cases. As shown in Table II the overhead ranges from a minimum of 9.34% of the *ADPCM* to a maximum of 212% of *FFT*. The worst results obtained with *FFT* and *DSPI* are due to two main reasons. First, the starting RTL and manual C/C++ models differ in some sub-functionality. That is, the manual C/C++ code implements a reduced set of features that, in contrast, are implemented in the RTL model and thus maintained in the C++ code generated by *H2C++*. For instance, the manual C/C++ code does not implement reconfigurability features (e.g., normalization power for *FFT* and transfer format for *DSPI*). This underlines the fact that, in many cases, it is hard to find an existing C/C++ fully compatible to the RTL model. The proposed methodology is more motivated when such a compatibility is a constraint.

Second, the number of asynchronous processes, the articulated protocol handshaking and, for *FFT*, the use of real types in conditional statements make both *FFT* and *DSPI* structurally hard to abstract.

In several other cases (i.e., *ADPCM*, *GCD*, *ECC*), the small difference between manual and automatically generated code is, in our opinion, reasonable.

### 8.1. Performance estimation and comparison with state of the art tools

To evaluate the performance of the code generated through the proposed approach, we compared the simulation time of the *H2C++* code with the code generated by the tools at the state of the art over the same sets of benchmarks (i.e., both classes 1 and 2) previously described. Among the tools reported in the related work section, we selected those whose license agreement has not constraints on performance comparison and publication. In particular, we selected *FreeHDL* [DIE.NET 2014] to convert VHDL to C/C++, while *Icarus* [Icarus 2014] and *Verilator* [VERILATOR 2014] to convert Verilog to C/C++ since they are the most robust and mature tools. Table III reports the results, which have been obtained on the same architecture and operating system previously described. The table reports whether the tools successfully converted the HDL code into C/C++ and whether the converted code is functionally equivalent to the starting HDL code. Performance comparison has been conducted only on the functionally equivalent code. *FreeHDL* parses VHDL code only, while *Icarus* and *Verilator* parse Verilog code only. *H2C++* parses both VHDL and Verilog.

The results show that *H2C++* outperforms the competitor in case of VHDL to C/C++ conversion, by generating code whose speedup ranges from 10.27x to 26.81x. This is due to the several optimizations *H2C++* applies to the C++ code, and, in particular, to the abstraction of data types and communication protocol.

In case of Verilog to C/C++ conversion, the results show that the competitor tools are as robust (i.e., they can parse complex Verilog implementations) as *H2C++*. Still, *H2C++* outperforms *Icarus* by generating code whose speedup ranges from 24.18x to 77.39x. In contrast, *Verilator* generated code that is comparable (in terms of simulation performance) to that generated by *H2C++* and, in some cases, slightly better. In par-

Table III. Performance evaluation and comparison with state of the art tools

CL	RTL IP block	HDL	Conversion success	RTL/C++ equiv.	Simulation time (ms)			
					FreeHDL	Icarus	Verilator	H2C++
1	LP_IIR_Filter	VHDL	Yes(FreeHDL)	Yes	24,379	-	-	1,341
	BM_Lambda	Verilog	Yes(Icarus, Verilator)	Yes	-	19,393	702	772
	Lambda_roots	Verilog	Yes(Icarus, Verilator)	Yes	-	6,494	191	239
	Omega_phy	Verilog	Yes(Icarus, Verilator)	Yes	-	70,847	2,093	2,930
	Out_stage	Verilog	Yes(Icarus, Verilator)	Yes	-	5,404	176	185
	Error_corr	Verilog	Yes(Icarus, Verilator)	Yes	-	28,593	1,009	1,039
	DIV_FRS	VHDL	No(FreeHDL)	-	-	-	-	1,551
	DIST_Fil_FRS	VHDL	No(FreeHDL)	-	-	-	-	1,635
2	JPEG	Verilog	Yes(Icarus, Verilator)	Yes	-	170,490	1,695	2,203
	FFT	VHDL	No(FreeHDL)	-	-	-	-	1,280
	DSPI	VHDL	Yes(FreeHDL)	No	-	-	-	3,870
	ADPCM	VHDL	Yes(FreeHDL)	No	-	-	-	4,330
	CRC	VHDL	No(FreeHDL)	-	-	-	-	5,360
	Root_FRS	VHDL	Yes(FreeHDL)	Yes	9,919	-	-	370
	GCD	VHDL	No(FreeHDL)	-	-	-	-	1,770
	ECC	VHDL	Yes(FreeHDL)	Yes	3,491	-	-	340

ticular, *Verilator* generated code whose speedup ranges from 1x to 1.33x w.r.t. *H2C++*. This is due to the fact that *Verilator*, besides abstracting data types, it generates very *low level C* code. On the one hand, this allows gaining in simulation performance, which is the main *Verilator* goal. On the other hand, the code loses readability and maintainability, which are, among others, goals of the proposed approach.

## 9. CONCLUSIONS

This article presented a methodology to automatically generate high-level C++ descriptions starting from existing RTL IPs. The methodology aims at abstracting the architectural details, which are typical of the HW implementations, to generate efficient C++ code to be compiled and executed as SW applications on MPSoC CPUs. The experimental results showed that the performance of the code automatically recovered from RTL descriptions are in general not better than the performance of the code manually implemented. However, the performance are similar in many cases. For these cases, the proposed approach allows designers to save manual work for re-implementing and, especially, for verifying such a code. The article presented an analysis to understand for which classes of RTL IPs the proposed methodology is more effective and the characteristics of the RTL descriptions that can influence the performance of the automatically generated C++ code. In HW/SW partitioning, this allows designers to understand whether there is room for (and thus it is worth) improving the SW application through a manual re-implementation rather than maintaining the IP as HW block.

## ACKNOWLEDGMENTS

The authors would like to thank Giovanni Auditore of STMicroelectronics and Romain Lemaire of CEA for providing the industrial set of benchmarks and supporting the analysis of the application results.

## REFERENCES

- ALDEC. 2014. HES-DVM. <http://aldec.com> (2014).
- Gasser Ayad, Andrea Acquaviva, Enrico Macii, Brahim Sahbi, and Romain Lemaire. 2013. HW-SW Integration for Energy-efficient/Variability-aware Computing. In *Proc. of ACM/IEEE Conference on Design, Automation and Test in Europe (DATE '13)*. 607–611.
- Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. 2010. Abstraction of RTL IPs into Embedded Software. In *ACM/IEEE Proceedings of the 47th Design Automation Conference (DAC '10)*. 24–29.
- Nicola Bombieri, Franco Fummi, Graziano Pravadelli, and Joao Marques-Silva. 2007. Towards Equivalence Checking Between TLM and RTL Models. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE '07)*. 113–122.
- Carbon Design Systems. 2014. Carbon Model Studio. <http://carbondesignsystems.com/> (2014).

- Kwang-Ting Cheng and A. S. Krishnakumar. 1996. Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1 (Jan. 1996), 57–79.
- DIE.NET. 2014. FreeHDL-V2CC. <http://linux.die.net/man/1/freehdl-v2cc> (2014).
- Wolfgang Ecker, Volkan Esen, Lars Schönberg, Thomas Steininger, Michael Velten, and Michael Hull. 2007. Interactive Presentation: Impact of Description Language, Abstraction Layer, and Value Representation on Simulation Performance. In *ACM/IEEE Proceedings of the Conference on Design, Automation and Test in Europe (DATE '07)*. 767–772.
- EDALAB. 2014. HIFSuite: Tools for HDL code conversion and manipulation. <http://hifsuite.edalab.it> (2014).
- Stephen A. Edwards. 2003. Tutorial: Compiling Concurrent Languages for Sequential Processors. *ACM Trans. Des. Autom. Electron. Syst.* 8, 2 (April 2003), 141–187.
- Freescale. 2009. Embedded multicore: an introduction. [http://www.freescale.com/files/32bit/doc/ref\\_manual/EMBMCRM.pdf](http://www.freescale.com/files/32bit/doc/ref_manual/EMBMCRM.pdf). (2009).
- Robert S. French, Monica S. Lam, Jeremy R. Levitt, and Kunle Olukotun. 1995. A General Method for Compiling Event-driven Simulations. In *Proc. of the 32Nd Annual ACM/IEEE Design Automation Conference (DAC '95)*. 151–156.
- Abdelaziz Guerrouat and Harald Richter. 2006. A component-based specification approach for embedded systems using FDTs. *ACM SIGSOFT Softw. Eng. Notes* 31, 2 (2006), 14–18.
- Craig Hansen. 1988. Hardware logic simulation by compilation. In *Proc. of the 25Nd Annual ACM/IEEE Design Automation Conference (DAC '88)*. 712–715.
- F. Herrera, H. Posadas, P. Sanchez, and E. Villar. 2003. Systematic Embedded Software Generation from SystemC. In *ACM/IEEE Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*. 142–147.
- Icarus. 2014. Icarus Verilog. <http://iverilog.icarus.com> (2014).
- ITRS. 2011. *International Technology Roadmap for Semiconductors - 2011*. <http://www.itrs.net/Links/2011ITRS/2011Chapters/2011SysDrivers.pdf>.
- Hisaaaki Katagiri, Keiichi Yasumoto, Akira Kitajima, Teruo Higashino, and Kenichi Taniguchi. 2000. Hardware Implementation of Communication Protocols Modeled by Concurrent EFSMs with Multi-way Synchronization. In *ACM/IEEE Proc. of Design Automation Conference (DAC '00)*. 762–767.
- T. John Koo, Bruno Sinopoli, Alberto Sangiovanni-Vincentelli, and Shankar Sastry. 1999. A Formal Approach to Reactive System Design: Unmanned Aerial Vehicle Flight Management System Design Example. In *IEEE Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*. 522–527.
- Grant Martin. 2006. Overview of the MPSoC Design Challenge. In *Proceedings of the 43rd Annual Design Automation Conference (DAC '06)*. 274–279.
- Mentor Graphics. 1994. Introduction to VHDL. [http://pages.cs.wisc.edu/sohi/cs552/Handouts/MentorDocs/mentor\\_graph\\_ics\\_introduction\\_to\\_VHDL.pdf](http://pages.cs.wisc.edu/sohi/cs552/Handouts/MentorDocs/mentor_graph_ics_introduction_to_VHDL.pdf). (July 1994).
- OSCI. 2002. Functional specification for SystemC 2.0. <http://www.systemc.org>. (April 2002).
- OSTATIC. 2014. VHDLC. <http://ostatic.com> (2014).
- Kalyan Saladi, Harikumar Somakumar, and Mahadevan Ganapathi. 2013. Concurrency-aware Compiler Optimizations for Hardware Description Languages. *ACM Trans. Des. Autom. Electron. Syst.* 18, 1, Article 10 (Jan. 2013), 10:1–10:16 pages.
- W. Stoye, D. Greaves, N. Richards, and J. Green. 2003. Using RTL-to-C++ Translation for Large SoC Concurrent Engineering: A Case Study. *IEEE Electronics Systems and Software* 1, 1 (2003), 20–25.
- IEEE SystemC. 2006. Standard SystemC Language Reference Manual. <http://ieeexplore.ieee.org>. (2006).
- TouchMore. 2013. Automatic Customizable Tool-Chain for Heterogeneous Multicore Platform Software Development (FP7 ICT-288166). <http://www.touchmore-project.eu>. (2013).
- VERILATOR. 2014. Verilator - Convert Verilog code to C++/SystemC. <http://www.veripool.org/wiki/verilator> (2014).
- IEEE Verilog. 2006. Standard for Verilog Hardware Description Language. <http://ieeexplore.ieee.org>. (2006).
- IEEE VHDL. 1994. Standard VHDL Language Reference Manual. <http://ieeexplore.ieee.org>. (1994).
- W. Wolf, A.A. Jerraya, and G. Martin. 2008. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 27, 10 (Oct 2008), 1701–1713.
- Abdelkrim Zitouni, Sami Badrouchi, and Rached Tourki. 2006. Communication Architecture Synthesis for Multi-bus SoC. *Journal of Computer Science* 2, 1 (2006), 63–71.