Using High-level Languages for Hardware Modeling and Implementation

Nelson Ferreira, Filipe Teixeira,
Nuno Lau, Arnaldo Oliveira, Orlando Moreira[1]

Abstract – **This paper describes the use of high-level languages in hardware modeling and implementation. The purpose of the article is to describe a methodology that can be used in the design of a new system. First we will describe the main phases of hardware design flow, namely: modeling, validation, synthesis, implementation, prototyping and testing. We will also give a brief overview of some high-level languages. Afterwards, we will propose a methodology, where a new system is designed using successively a subset of C++, SystemC and VHDL using some guidelines to provide a smooth transition between languages and levels of abstraction. We will present a case study where an UART has been designed using this methodology. We will report the advantages and disadvantages of each language. This methodology provided a clear refinement flow from a functional sequential model to a RTL synthesizable model, although it created some consistency problems. The UART was implemented together with a MIPS32 processor within a FPGA for prototyping and testing purposes.**

Resumo – **Este artigo descreve a utilização de linguagens de alto nível na modelação e implementação de hardware. O objectivo deste artigo é apresentar uma metodologia que pode ser usada no projecto de novos modelos de sistemas. Primeiro iremos descrever as principais fases no fluxo de projecto de hardware, nomeadamente: modelação, validação, síntese, implementação, prototipagem e teste. Também iremos apresentar uma breve descrição de algumas linguagem de alto nível. Posteriormente, iremos propor uma metodologia usando algumas regras que permitem obter uma transição suave entre diferentes linguagens e níveis de abstracção, quando um sistema é modelado usando sequencialmente um subconjunto das linguagens C++, SystemC e VHDL nas diferentes fases do projecto. Será apresentado um *case study* do projecto de uma UART utilizando a metodologia proposta. Iremos expor as vantagens e desvantagens de cada linguagem. Esta metodologia permitiu obter uma passagem suave do modelo funcional até ao modelo RTL sintetizável, no entanto criou alguns problemas de inconsistência. A UART foi implementada para teste e propotipagem conjuntamente com um processador MIPS32.**

Keywords – **System Specification, Hardware Design Flow, Modeling, Synthesis, VHDL, SystemC, FPGA Prototyping, UART Design**

Palavras chave – **Especificação de um sistema, Fluxo de projecto de hardware, Modelação, Síntese, VHDL, SystemC, Prototipagem em FPGA, Projecto de uma UART**

I. Introduction

The first design stage of a digital system is the specification of its global functionality, the definition of the main interfaces and all other relevant characteristics and constraints. When designing a trivial system, natural languages are often used to build the respective specification. However, with the increase of system complexity, formal specifications are preferred because they can be verified, analyzed, simulated and synthesized with Computer Aided Design (CAD) tools.

Ideally, for complex systems, the specification must be the first step of a well defined design methodology. Models can be produced in a variety of high-level languages, such as C based languages or Hardware Description Languages (HDLs). Software engineers prefer software programming languages that provide a great level of abstraction. On the other hand, hardware engineers prefer HDLs, that provide adequate abstractions for hardware modeling. Furthermore when designing a system there is always the question of what should be implemented in software and what should be implemented in hardware. The ideal would be a tool that could, from a description in system level modeling language, separate what to implement in software and what to implement in hardware. Currently such tools are relatively immature, not widely available and mainly application domain specific [1].

After creating a model the developer must consider the validation, the synthesis, the implementation, the prototyping and finally the tests.

There are many methods to specify a digital system, namely boolean equations, schematic diagrams, graphical languages, HDLs, and system level modeling languages, depending on the abstraction level. We propose in this paper a methodology that starts by modeling the system in a C++ subset, and refines the model using SystemC [2] and VHDL [3]. This work has the following objectives:

[1]Philips Research Laboratories - Eindhoven

- Understand the suitability of different languages to each design phase.
- Establish the guidelines required to provide a smooth transition between different phases in the design flow using different languages.
- Create a methodology that implements those guidelines.
- Evaluate the methodology using a real world example.

This paper contains four more sections. In section II we explain the major steps in hardware development detailing the languages and the requirements of each one. In section III we present the proposed methodology. Section IV presents the development of an UART as a case study of the proposed methodology and present the results and the discussion. Finally, in section V we draw the conclusions.

II. HARDWARE DEVELOPMENT

A. Modeling

The result of specification is a model, i.e., a representation that shows the relevant characteristics without the associated details. It must incorporate all functional characteristics of the system without considering any implementation details such as specific components used or particular hardware/software partitions of the system implementation. Functional models are often used, at early design stages, to validate the algorithm that is going to be used in the system. The models can be produced in a variety of high-level languages, but in this case we used languages based on C and HDLs. In the section II-A.1 we describe some of these languages.

A.1 High-level Languages

C based languages

The C++ language is used in complex systems to write an executable specification and to develop software. The great advantage of the use of C++ for hardware modeling comes from its wide adoption and large programmers base. However, C++, in its original form, has some limitations to be used for hardware specification namely:

- Lack of appropriate data types;
- Absence of concurrency, reactivity and notion time;
- Great degree of freedom.

These can be considered the disadvantages of the use of C++ for hardware specification.

The SystemC [2] language was created to overcome the limitations of C++ in hardware modeling. SystemC is a set of class libraries implemented on top of C++, that supports hardware modeling concepts like concurrency, reactivity and latency. SystemC provides constructs that describe concepts that are familiar to hardware designers such as signals, modules and ports. In other words with SystemC we can define hardware and software components. SystemC also provides a simulation kernel that allows the designer to simulate the executable specification using just an ordinary C++ compiler. With SystemC the step-by-step refinement of a system design down to the RTL for synthesis is simplified.

Hardware Description Languages

VHDL [3] is a hardware description language where we can describe the model of an hardware system. The goal of VHDL creators was to create an unambiguous, portable and general language. However, because VHDL is a strongly typed language, the syntax of VHDL becomes verbose (e.g. additional code is often needed to explicitly convert one data type to another). On the other hand, the strongly type feature can be beneficial to detect errors at early design stages.

B. Validation

The validation of a model can be made in two different ways: formal verification and validation through simulation. In the design of a system the validation must be executed at several stages in order to validate the results obtained in the stages that precede it and to detect errors as soon as possible.

The validation through simulation is the most used when it is intended to make a validation of a system. The objective is to execute a logical verification and make an analysis of the performance. Generally, the use of testbenches is convenient to execute a logical verification. They apply stimulus to the system inputs. This type of verification has the disadvantage of being insufficient for complex systems, because it isn't an exhaustive method. In most complex projects it is impossible for test vectors to cover all the cases.

The formal verification is based on the use of mathematical methods to verify the functionality of the system. It has the advantage of not needing test vectors and supplies an exhausting verification. It can be used as a complement of the previous method.

C. Synthesis and Implementation

The synthesis is the process used to obtain, from the behaviorial description, a structural description in a lower abstraction level. Generally the synthesis is the partition in modules of the behavioral description. Depending on the abstraction level we can have different kinds of synthesis, such as system synthesis and logic synthesis. With system synthesis it is possible to decompose an abstract specification of the system in a software implementation and a hardware implementation. One objective of the system synthesis can be the reuse of predefined components, e.g., *Intellectual Property* blocks. Logic synthesis generates the circuit that implements a given logic specification, e.g., generate circuits with finite state machines through connection of flip-flops. Synthesis can be realized manually, or automatically using CAD tools, although the manual

process is slow and error prone. The final result of synthesis is a circuit described in a the form of a netlist.

Finally, the implementation is the process of mapping, placement and routing. Mapping is the adaptation of our netlist to the components available in a given technology. Placement is the positioning of each component in the available area of the implementation. Routing determines the path of the signals that connect the component interfaces.

D. Prototyping and Testing

After implementation of the design the developer comes across with the necessity to foresee the real behavior of its design.

Prototyping consists in the creation of a functioning version of the final system without some of the deployment characteristics. Some constraints related to area, cost and power consumption may be relaxed in this phase.

III. Proposed Methodology

Our methodology proposes the guidelines required to provide a smooth transition from behavioral abstraction level to RTL synthetisable level using different languages at different stages. It uses C++ to create the executable specification, then SystemC to create the hardware/software models and finally VHDL to obtain a synthetisable model. C++ was chosen due to its encapsulation capabilities and because it is the SystemC base language. In figure 1 we present a diagram with the transition between the different stages.

We start by defining the specification of the system, i.e, the interfaces and the behavior. To guarantee a smooth transition, we propose some coding guidelines, that must be taken into account while building the functional model:

- Each component must be implemented as a C++ class.
- Use few data types and avoid the use of pointers.
- Hardware ports are modeled as class constructor parameters and port bindings are implemented using external shared variables.
- Asynchronous and synchronous events are implemented as independent functions.
- Because of the lack of concurrency support, the order of function invocation must be considered, for a correct simulation progress.

If these guidelines are followed the transition to SystemC is smooth and with very little changes from the C++ model. The functions used in the C++ model to simulate synchronous and asynchronous events are translated to signals of SystemC method sensitivity list. The use of HDLs at the final stage of development is justified by the fact that the synthesis tools are more developed for these languages, the synthetisable language subset is well documented and for its strong checking capabilities.

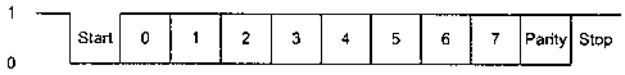

Figure 2 - Frame format

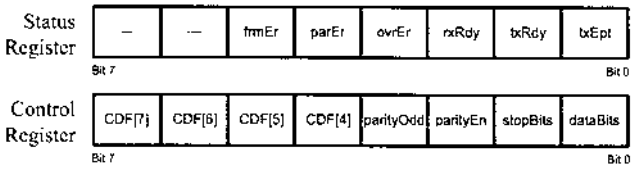

Figure 3 - UART Control and Status Registers

The innovation of this methodology isn't the successive use of C++, SystemC and VHDL at the different design stages [4] [5] but it is the application of the guidelines that provide a smooth transition from behavioral abstraction level to hardware synthesis.

IV. A Real World Example

A. The RS232 Protocol

The RS232 Protocol is an asynchronous serial communication [6] method used in point to point interfaces. The protocol describes a communication method where transactions of information are made character by character. In an asynchronous communication link there is no separate clock line, thus the data must be synchronized using others methods, such as special synchronization bits within the data frame.

Another important consideration is the transmission baud rate. The transmitter and receiver must be programmed to use the same bit frequency.

Figure 2 shows the frame in RS232 Protocol composed by a *Start Bit*, *Data Bits*, *Parity Bit* and *Stop Bits*. The *Start Bit* is used for synchronization purposes. Because the line is in marking state (on state) when idle, the *Start Bit* (off state) is easily recognized by the receiver. The *Data Bits* are sent immediately following the *Start Bit* and the least significant bit is always the first bit sent. The *Parity Bit* exists for detecting errors. The parity can be calculated in Even parity or in Odd parity. The *Stop Bits* identify the end of a data frame, in other words we can say that the stop bit is the minimal interval between two consecutive characters.

B. UART Interface

An UART is a hardware component used to implement RS232 protocol. Our implementation provides a synchronous memory-like bus interface for applications that communicate over RS232. The UART has a control register where the transmission parameters can be defined and a status register where some flags are stored. Besides the control and the status register the UART also has two data registers: a transmission buffer and a reception buffer. All registers are eight bits wide. The status register (see figure 3) has information about the status of transmission and reception buffers and also if errors have been detected. As we can see in figure 3 the status register is composed by:

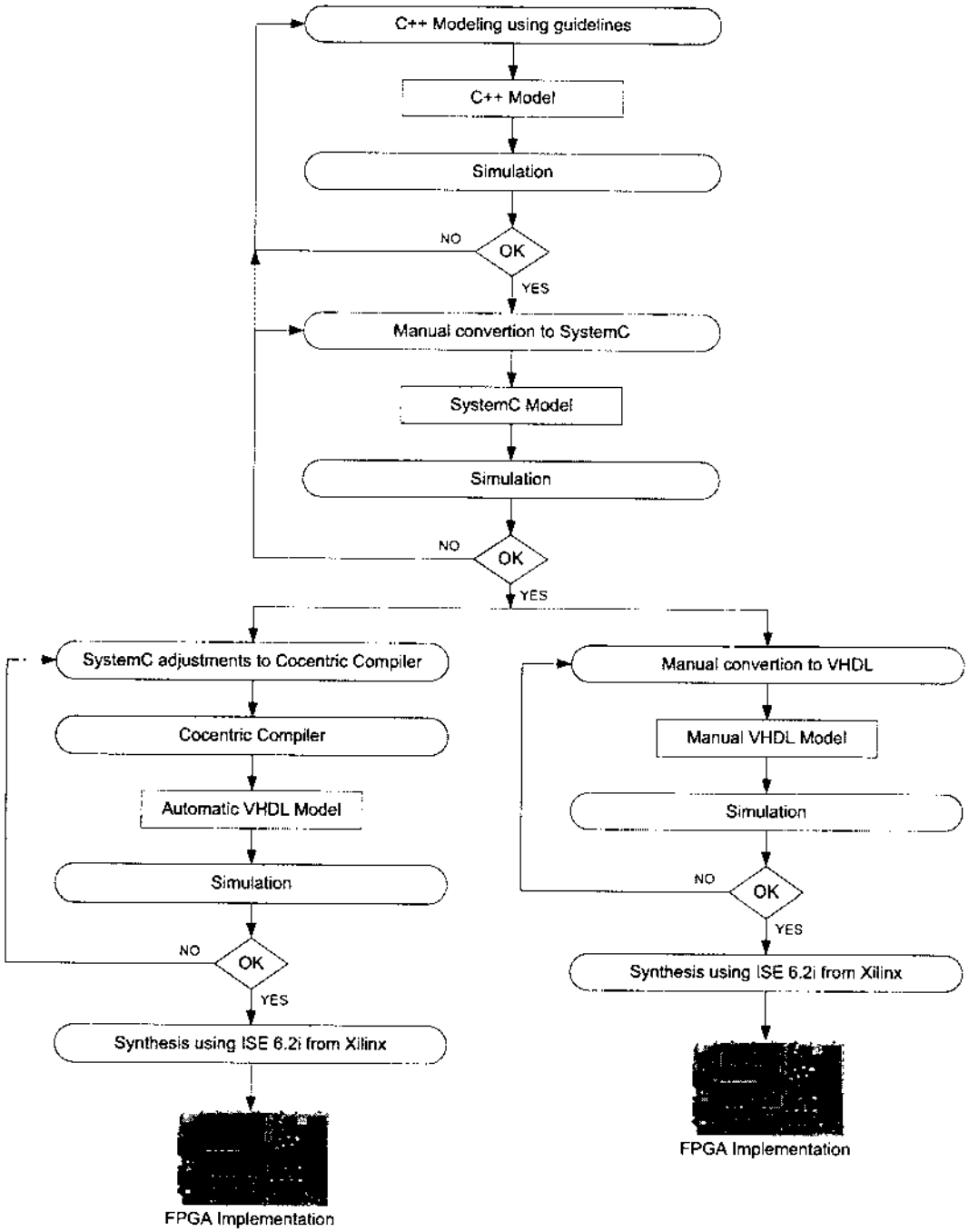

Figure 1 - Methodology diagram

frmEr – is set to "1" to indicate an invalid *Stop Bits* field due to noise, synchronization errors or configuration mismatches.

parEr – this bit is set to "1" to indicate that there is a parity error.

ovrEr – this bit is set to "1" when new data has been received and the previous data has not been read from reception buffer.

rxRdy – this bit is set to "1" to indicate that reception buffer has received a new character.

txRdy – this bit is set to "1" to indicate that the transmission buffer is ready to accept a new character.

txEpt – this bit is set to "1" to indicate that there is no character in the transmitting shift register.

The control register (see figure 3) is a programable register where the user can define some parameters such as:

CDF – these 4 bits define the value of the clock divider.

parityOdd – define the parity (odd or even).

parityEn – determines the use (or not) of a parity bit.

stopBits – defines the number of stop bits. The bit set to "1" indicates 2 stop bits, otherwise 1 stop bit is used.

dataBits – defines the number of data bits to transmit. The bit set to "1" indicates 8 data bits, otherwise

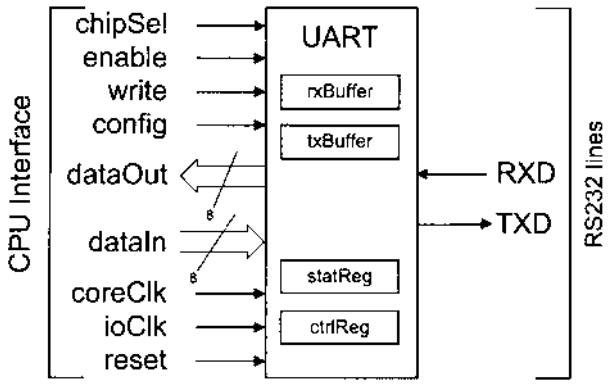

Figure 4 - UART Schematic

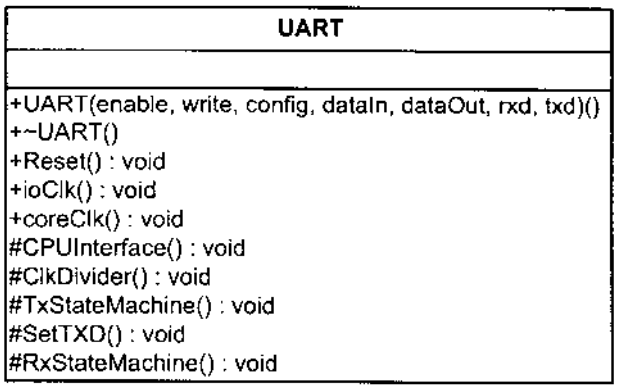

Figure 5 - CLASS UART

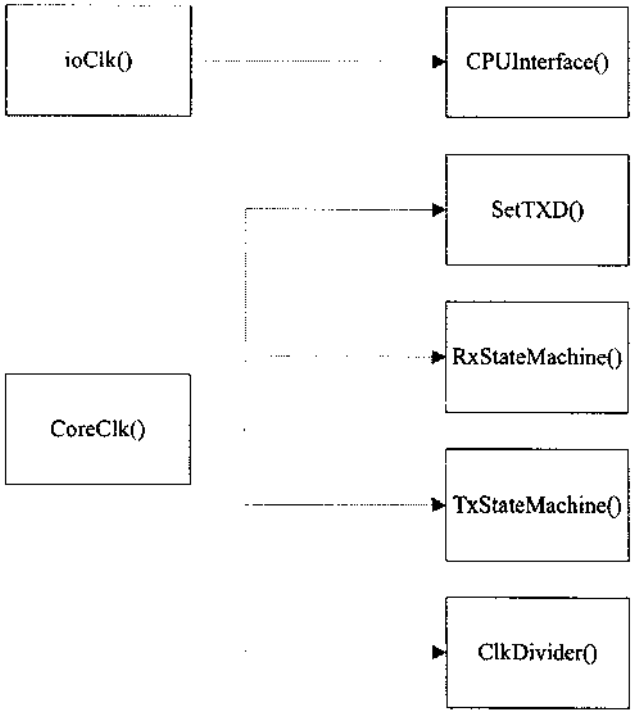

Figure 6 - Functions dependency

7 data bits are used.

In figure 4, we can see the ports that provide the interface with CPU, and the interface with the RS232 line. The *dataIn* and *dataOut* are eight bits wide buses, the remaining ports are used for control purposes. The values at each port are shown in table I. The interface with protocol is made with the lines *RXD* and *TXD*.

C. UART Specification

Before starting to build any model we have to define precisely what we pretend to obtain in the end. With this in mind we defined the specifications of our model, these specifications are done without the details of the implementation. We only describe what we pretend to obtain.

With this in mind we chosed to have two clock signals, one clock implements the synchronous interface between the CPU and the UART (*ioClk*), the second one is used to obtain the baud rates necessary for transmission and reception using a clock divider (*coreClk*).

To set the baud rates we use the four MSB of control register. In practise we can define values between 16 and 192 for clock divider then the baud rates varied between 9600 and 115200 for a frequency of 1.843Mhz to *coreClk*.

The UART must be able to transmit and receive at the same time, i.e., full duplex communication must be provided.

C.1 C++

C.2 Modeling

The modeling in C++ consisted in the construction of a class (figure 5) that allowed us to obtain a functional model of the UART.

The data types used to specify the variables were `bool` for single bit variables and `unsigned char` for eight bit variables. Regarding interface ports access, the shared variables that model the ports are passed as arguments to the class constructors (figure 5). The same shared variable can be used to connect multiple components, e.g., these variables may be shared between the CPU and the UART or between two UARTs. In the definition of the constructor we declare the parameters that are used to simulate the input ports of the UART interface as `const`, this way functions inside the class cannot set values to these parameters. Because C++ doesn't support the notion of time, we had to implement two functions, `ioClk()` and `coreClk()`, that simulate the synchronous events generated by the clocks. In hardware the functions that are triggered by an event are executed in parallel and in C++ they are executed sequentially, hence we had to take some cautions in the order of invoking the functions that implement the functionality of the design.

The functions that implement both clocks where subdivided in others functions as shown in figure 6, these functions are responsible for the implementation of the UART internal operations.

Function `CPUInterface()` implements the synchronous communication between the CPU and the UART.

	Operation						
Signals	Initialization	No Operation		WR Ctrl Reg	RD Stat Reg	WR Tx Buf	RD Rx Buf
reset	1	0	0	0	0	0	0
chipSel	x	0	x	1	1	1	1
enable	x	x	0	1	1	1	1
config	x	x	x	1	1	0	0
write	x	x	x	1	0	1	0
dataIn	xxxxxxxx	xxxxxxxx	xxxxxxxx	ctrl reg val	xxxxxxxx	char to tx	xxxxxxxx
dataOut	zzzzzzzz	zzzzzzzz	zzzzzzzz	zzzzzzzz	stat reg val	zzzzzzzz	rx char

Table I
PORT VALUES FOR EACH UART INTERFACE OPERATION

```
void UART::CPUInterface() {
    if ((m_enable == 1) && (m_chipSel == 1)) {
        if (m_config == 0) {
            if (m_write == 1){
                m_txBuffer = m_dataIn;
                m_statReg.m_bits.txRdy = 0;
            } else {
                m_dataOut = m_rxBuffer;
                m_statReg.m_bits.rxRdy = 0;
            }
        } else {
            if (m_write == 1) {
                m_ctrlReg.m_byte = m_dataIn;
            } else {
                m_dataOut = m_statReg.m_byte;
                m_statReg.m_bits.ovrEr = 0;
                m_statReg.m_bits.parEr = 0;
                m_statReg.m_bits.frmEr = 0;
            }
        }
    }
}
```

Figure 7 - C++ implementation of the function CPUInterface()

To provide the full duplex capability, we had to have two separate modules, one for transmission and one for reception. The module that implements the transmission is composed by two functions. `TxStateMachine()`, represents the transmitting state machine. The function `SetTxd()`, implements a multiplexer that, using the information provided by `TxStateMachine()`, drives the *TXD* line.

The module that implements the reception is composed by one function, `RxStateMachine()`, this function represents a state machine that is in charge with the reception of a character.

It is also necessary to implement a clock divider. Function `ClkDivider()` generates the standard baud rates that are used in the protocol communication.

The asynchronous reset interface is modeled with the independent function `Reset()`, this function initializes the internal variables of the UART.

To demonstrate the different implementations and changes in the languages used in our case study we will use the function `CPUInterface()`. In the C++ design the function has the code showed in figure 7.

This function implements a synchronous interface between the CPU and the UART, four operations can be performed by the CPU on the UART:

- Write to the control register
- Write to the transmission buffer
- Read from the status register
- Read from the reception buffer

The signals *enable* and *chipScl*, activate the CPU interface. When they are active, the signals *write* and *config* are tested to select the operation to perform. Depending on the operation, this function also performs the reset of some status bits.

C.3 Simulation

The simulation in C++ was based on the communication between two UARTs (figure 8). UART1 is responsible for transmitting the character pressed on the keyboard to UART2 and UART2 is responsible for the reception of the character and for sending it to the display. With this simple simulation we test both transmission and reception of the UART. To implement this simulation we created a `main` function that performs the following tasks: writes into the UART1 transmission buffer all characters pressed in the keyboard, polls the UART2 to detect new received characters and sends them to the display. The UART interface signals are driven accordingly to the operation performed.

We also created a class that manipulates Value Change Dump (VCD) files. This class was responsible to create and maintain one VCD file that stores the evolution in time of variables values. To visualize these files we used the application GTKWave [7]. This simulation allowed us to test the external interface of the UART, using the functional testbench, and the internal details, using the VCD file.

D. SystemC

D.1 Modeling

The transition from C++ to SystemC was done with very little changes in terms of the code to implement the functions. Changes occurred mainly in the declaration of the functions and the type of variables. In

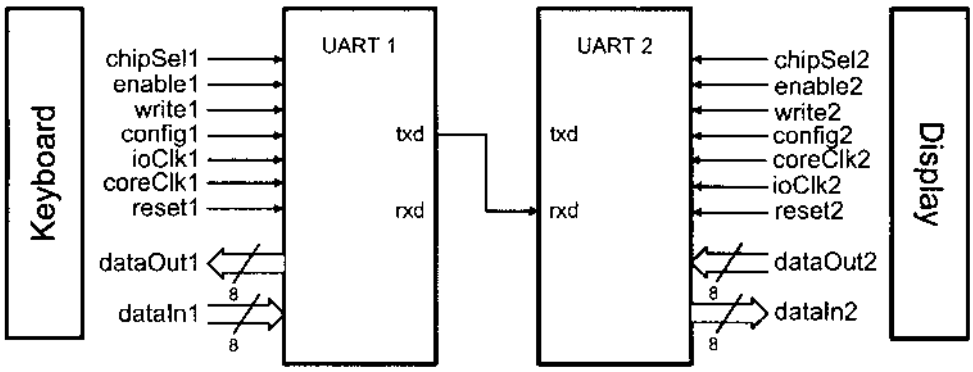

Figure 8 - Simulation setup

SystemC the UART class became a `SC_MODULE`, and some of the functions were implemented as `SC_METHODs` with a sensitive list of the signals that can trigger the method.

Unlike C++, SystemC provides concurrency and reactive behavior, so the functions that implemented the clocks, and the asynchronous events are no longer necessary. Event handling is transparently managed by the SystemC simulation kernel using the sensitivity list of the methods. Hence, in the SystemC model we don't have the function `Reset()`, all methods are sensitive to the signal *reset* and they perform the reset to the signals that they are responsible to drive. The addition of the reset capability to the former C++ functions implied some modifications in the code.

All the variables were converted to SystemC data types, variables declared as `unsigned char` were declared as `sc_lv<8>` or as `sc_uint`. All variables were declared as `sc_signal`.

The code showed in figure 9 implements the CPUInterface method using SystemC:

The modifications from C++, are mainly related with the integration of the reset functionality in the model. The other modifications are only related with the access to the SystemC data types.

```
void  UART::CPUInterface() {
 if (reset == 1) {
    s_ctrlReg = 0xF0;
 } else {
    if ( (ioClk.event())
         && (ioClk.read() == 1)) {
       if ((enable.read() == 1)
           && (chipSel.read() == 1)) {
          if (config.read() == 0) {
             if (write.read() == 1) {
                 s_txBuffer = dataIn.read();
                 s_statReg[1] = 0; }
             else {
                 dataOut.write(s_rxBuffer);
                 s_statReg[2] = 0; }
          } else {
            if (write.read() == 1) {
                s_ctrlReg = dataIn.read();
                sc_lv<8> aux = dataIn.read().range(7,4);
                s_clkDivFactor.range(7,4) = aux; }
             else {
                dataOut.write(s_statReg);
                s_statReg = (s_statReg & 0x07); }
          }
       }
    }
 }
}
```

Figure 9 - SystemC implementation of the function CPUInterface()

D.2 Simulation

The simulation in SystemC was done with a testbench. To implement it we created a new module. The testbench module was specified with a process of type `SC_THREAD`. In that process we provided the values that must be assigned to the interface of the UART in a sequential form and after each modification we add the function `wait()`.

Functions to manipulate VCD files are already supplied by the SystemC library. To visualize the files produced we used the application GTKWave.

D.3 Synthesis and Implementation

To perform the SystemC synthesis we used the aplication CoCentric SystemC Compiler [8].

Although the code produced so far simulated correctly, to be able to convert it to VHDL using the CoCentric compiler we had to make some changes because the application has the following restrictions:

- Does not support multi-source signals, i.e., although signals can be read by multiple methods they can only be changed by one method.
- For single bit variables the use of the data type `bool` is advised.
- Does not support the function `event()` inside the methods to test witch signal generated the event.
- Sensitivity lists that mix level and edge sensitivities are not allowed.

The signal that was causing multi-source problems was the status register, because we had to change its bits in the methods used for the state machines and in the method that implemented `CPUInterface()`. The bits that were causing multi-source problems were the *txRdy* because it is set in the method `TxStateMachine()` and is reset in the

	VHDL by Cocentric	Manual VHDL
Max. Frequency	55.408Mhz	55.408Mhz
# slices used	152 out of 3072	122 out of 3072
# slice Flip Flops	112 out of 6144	102 out of 6144

Table II
DEVICE (SPARTAN II-XC2S300E) UTILIZATION OBTAINED USING SYNTHESIS TOOLS

method `CPUInterface()`, the bit *rxRdy* because it is set in the method `RxStateMachine()` and is reset in `CPUInterface()`, and the bits *ovrEr*, *frmEr* and *parEr* because they are set in the method `RxStateMachine()` and the reset in `CPUInterface()`.

The solution adopted consisted in creating additional methods that generate a reset signal that is added to the sensitive list of the methods `RxStateMachine()` and `TxStateMachine()`, this way only the later methods change the bits from the Status Register.

After resolving these problems we were able to obtain a VHDL code, the code was synthesized using the application Xilinx ISE 6 and the Spartan II-E FPGA as target device. The results are presented in table II.

E. VHDL

In order to assess the quality of the VHDL code produced by the CoCentric SystemC Compiler, we performed the manual translation of the SystemC code into VHDL. The hand coded VHDL was then synthesized and the results were compared. This translation also allowed us to evaluate the involved effort, validating the proposed methodology.

E.1 Modeling

The manual conversion to VHDL consisted in transforming the SystemC methods into processes, convert the data types to VHDL data types and make the necessary changes in the syntax of the languages.

The VHDL code that implements the `CPUInterface` process is showed in figure 10.

E.2 Simulation

The simulation in VHDL was made using ModelSim® with a testbench that is similar to the one implemented in SystemC with correct adjustments to the VHDL language.

E.3 Synthesis and Implementation

The VHDL module was synthesized with the application Xilinx ISE 6 [9] to be used in the device Spartan II-E FPGA, the synthesis generated the device utilization summary presented in table II.

F. Comparing Methods

F.1 Modeling

Comparing the three methods we used we can conclude that C++ is certainly adequate for modeling the

```
cpu_interface : process(reset, ioClk) begin
    if (reset = '1') then
        s_ctrlReg <= "11110000";
    elsif (rising_edge(ioClk)) then
        if ((chipSel = '1') and (enable = '1')) then
            if (config = '0') then
                if (write = '1') then
                    s_txBuffer  <= dataIn;
                else
                    s_dataOut   <= s_rxBuffer;
                end if;

            else
                if (write = '1') then
                    s_ctrlReg   <= dataIn;
                else
                    s_dataOut   <= s_statReg;
                end if;
            end if;
        end if;
    end if;
end process;
```

Figure 10 - VHDL implementation of the function CPUInterface()

software components of a system and to test the algorithm, but when it comes to modeling the hardware behavior it lacks some functionality.

SystemC, even if it is based on C++, has some functionalities that are more hardware like, and it benefits from a higher level of abstraction than conventional VHDL design.

The VHDL model is the most close to the hardware specification, however we think that the constraints imposed by the hardware description languages would be inconvenient in the early development stages. In our opinion the quality of the final VHDL code resulting from the proposed methodology beneficiated from the iterative refinement process.

F.2 Simulation

The simulation in C++ is more efficient at early stages of development to validate the algorithms and the sequence of operations without worrying about clocks and precise timings. It also helps in the functional decomposition of the system.

SystemC provides the concepts of concurrency and parallel execution of processes. Using SystemC modules run conceptually in parallel which may allow the detection of hidden data dependencies that were not visible in the C++ simulation. SystemC directly supports the generation of VCD files, which facilitates the visualization of the simulation results.

Two models of the same circuit, one written in SystemC and the other in VHDL must produce the same simulation output. However, the simulation of the SystemC model can be done with the SystemC library and a standard C++ compiler, which are both freely available. On the other hand the simulation in VHDL requires a specific simulator.

`void UARTSetup (TBaudRate baudRate, TDataBits dataBits, TParity parity, TStopBits stopBits);` This function programs the transmission parameters of the UART.
`void UARTTxChr (char chr);` This function sends a character to the UART to be transmitted.
`bool UARTRxChr (char* chr);` This function reads a received character from the UART.
`void UARTTxStr (const char* str);` This functions sends a string to the UART to be transmitted.

Table III
UART API DESCRIPTION

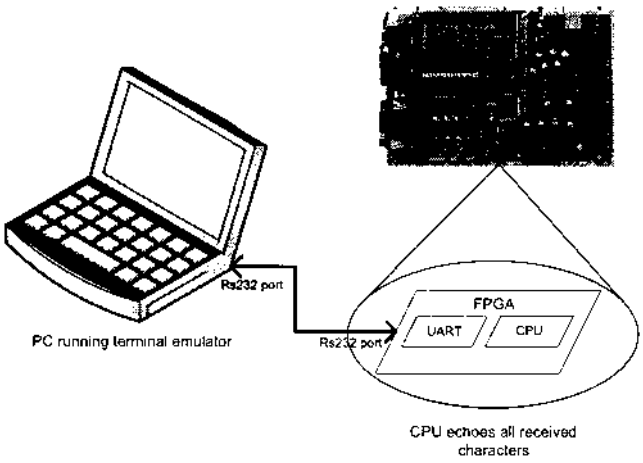

Figure 11 - Test setup

F.3 Synthesis and Implementation

The synthesis of the UART was made in two different ways: directly from the SystemC code using the Cocentric SystemC compiler and from hand-coded VHDL model using the Xilinx XST synthesis engine incorporated within Xilinx ISE Design environment.

The Cocentric SystemC compiler constraints force a non natural coding style. To obtain the SystemC synthesizable code we had to analyze how different SystemC constructions were synthesized in order to make the compiler generate synthesis friendly VHDL code.

The results of both synthesis flows are summarized into table II. Comparing results we can conclude that direct VHDL synthesis provides better results in terms of area.

G. Testing

The test of the UART was made in a FPGA (Development Board model TE-XC2SE from Trenz [10]) integrated with a MIPS32 processor [11].

The test consisted in the simulation of the communication between two computers. The program implemented in the MIPS32 just echoes all the characters received by the UART back to the computer. Using a terminal emulator we established a communication between the computer and the development board as seen in figure 11.

G.1 The UART API

To test the UART with the MIPS32 processor, we had to create an interface library. The library is described in table III.

V. CONCLUSION

The discussion on the utilization of C++, SystemC and VHDL at different stages of development presents a clear view of the advantages and disadvantages of each language.

The proposed methodology does not solve the consistency problems when translating the model manually. We also came across with the lack of support for this methodology of some of the tools that were used.

With our example we have shown that it is possible to have a smooth transition from the various phases of designing a system, using different design languages, when we follow the proposed guidelines.

REFERENCES

[1] Wayne Wolf, "A decade of hardware/software codesign", *IEEE Computer*, vol. 36, no. 4, pp. 38–43, April 2003.

[2] *Open SystemC Initiative*, http://www.systemc.org.

[3] IEEE, *IEEE Standard VHDL Language Reference Manual*, 2000 edition.

[4] Eike Grimpe and Frank Oppenheimer, "Extending the systemc synthesis subset by object-oriented features", in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2003, pp. 25–30, ACM Press.

[5] Yuval Ronen J.R. Armstrong, "Modeling with systemc: A case study", 2001.

[6] Electronic Industries Association, *EIA232E - Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*, 1991.

[7] *GTKWave Homepage*, http://www.linuxworkshop.com/bybell/ver/wave.

[8] *Describing Synthesizable RTL in SystemC, Version 1.2, November 2002*, http://www.synopsys.com.

[9] *Xilinx, inc.* http://www.xilinx.com.

[10] Trenz Electronic, http://www.trenz-electronic.de, *Spartan-IIE Development Platform Overview*, 2004.

[11] Arnaldo S. R. Oliveira António B. Ferrari , Valery A. Sklyarov, "Arpa - an open source system-on-chip for real-time applications", ERTSI - Embedded Real-Time Systems Implementation Workshop, 2004.