A SystemC-based Verification Methodology for Complex Wireless Software IP

Guido Post, P.K.Venkataraghavan, Tapan Ray, D.R.Seetharaman
Solutions Group, Synopsys Inc, {post, pkvenkat, tap, drsraman}@synopsys.com

Abstract

The implementation of a complex hardware Intellectual Property (IP) together with complex lower-level software and the integration into a system platform poses tough challenges to the design and verification engineers. Traditionally, embedded software is developed and tested towards the end of the development cycle because of late availability of lab prototype equipment and hardware IP. In this paper, a "software-centric" hardware/software implementation and verification methodology for a 3G WCDMA modem is presented, with emphasis on physical layer software design and early verification. The sub-system architecture of 3G hardware and software is presented along with design and verification steps carried out. A versatile SystemC-based test environment is described, which links test case modules producing the stimuli from protocol stack and hardware components to the L1 SW code, executed on a instruction set simulator.

1. Introduction

Currently, third generation (3G) mobile systems are being designed for high quality and high data rate services. For end-user equipment these systems are most often designed as IP blocks, which are then integrated together with other IP blocks and subsystems, such as RF front-end, connectivity IP (Bluetooth, UARTs), as well as software IP blocks (protocol stack, applications). In this paper, we focus on software (SW) and hardware (HW) IP creation for complex mobile subsystem, such as a physical layer (L1) SW and base-band processing HW for a Wideband CDMA (WCDMA) user equipment modem (UE) for UMTS [1].

Figure 1 depicts the top-level architecture of a UMTS UE containing the L1 modem HW and L1 SW IP blocks as well as other components from an "IP-creator" viewpoint. It also indicates the heterogeneous nature of interfaces of this subsystem. One can identify HW/HW interfaces (e.g. between RF front-end and L1-HW), HW/SW interfaces (e.g. interface to the µP) and SW/SW interfaces (e.g. interface to the protocol stack).

The methodology described in this paper addresses the following design and verification issues in creation of an L1 SW IP:

- Complex interfaces to higher layers as well as L1 HW
- Verification of L1 SW early in the development cycle
- Verification of integrated L1 HW, SW and protocol stack

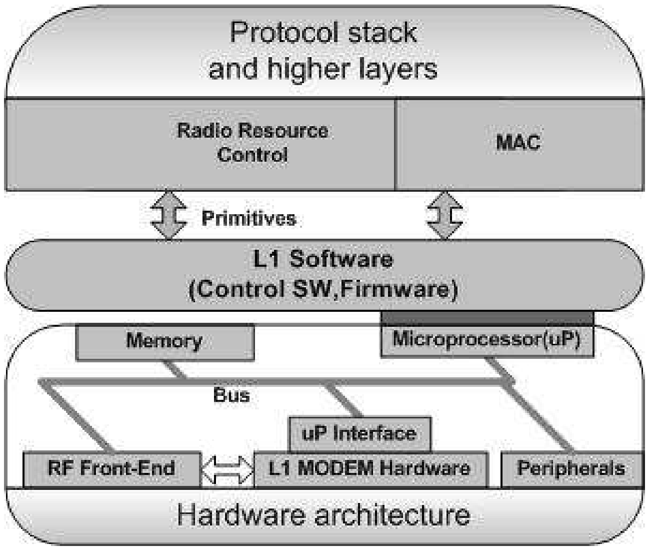

Figure1: To-level architecture of a UMTS UE

In a conventional design and developmental flow, the verification of L1 SW, which would be typically executed in embedded environment, is done once the HW prototype is available. This increases the development cycle time for complex wireless systems. Recently, advances have been made for incorporating cycle-callable simulation models of the µP architecture in a system-level environment delivering speed up to 100 kHz [2][3][4][5]. The simulation technology leverages a system level design, which is based on transaction-level modeling paradigms [6]. This is currently seen as the key enabler using SystemC to design systems in a more "software-centric" way[7][11]. But these simulation technologies do not cover the application-specific verification requirements. A verification methodology is required that accounts for the heterogeneity of the interfaces of the system lower-level embedded control SW and firmware.

The "software-centric" implementation and verification approach presented in this paper enables early verification of L1 SW in development cycle. The novelty of solution is based in the adaptation of a Testing and Test Control Notation (TTCN)[13] for describing software tests and integrating this notation on top of the SystemC. In addition, this approach provides the means for capturing the stimuli and responses from SW/SW and HW/SW interfaces to enable verification of integrated L1 HW, SW and protocol stack. The HW models required for testing the L1 SW are captured at different abstraction levels (such as high level abstraction for early verification and cycle true models for integrated system verification). The proposed platform, executed in CoCentric^{TM} System Studio (CCSS), includes:

- Test framework - Versatile SW verification library based on SystemC, in which test-cases are implemented and described
- Interfaces for integration of L1 SW with test framework
- L1 HW models at different abstraction levels

The overall design and verification flow, architecture and the design flow of L1 SW and "software-centric" verification platform are described in the next section. The subsequent section deals with HW/SW and HW/HW interface synthesis followed by concluding remarks.

2. Design and Verification Flow

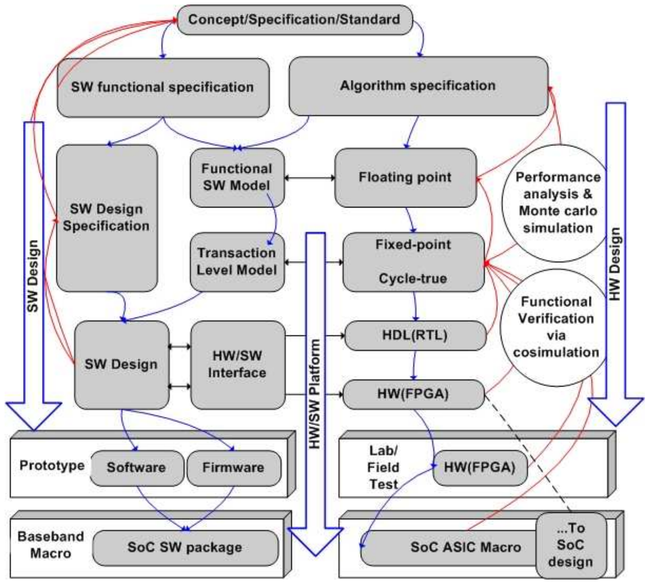

Figure2: Design and Verification Flow

Figure 2 depicts the overall design cycle from concept to silicon/prototype that starts with either system concept or standardized specification as in current case. From this system specification, further refinement is done to derive high-level SW functional specification and algorithm specification. Once the algorithms are validated, HW/SW partitioning is decided based on design requirements such as processing speed, low power and to minimize interactions between HW and SW. For example, in this case study, time critical tasks that are to be performed on a slot basis are implemented in HW, whereas tasks that are to be accomplished on a frame basis are done in SW. HW design and verification flow is described in [8]. The design and verification flow for SW implementation is elaborated in the following sub-sections with our case study as an example.

2.1 SW Design flow – L1 SW

Implementing complex software requires a thorough requirement analysis to arrive at a detailed functional and design specification. We used standard software engineering methodologies based on languages such as Unified Modeling Language (UML) [9], and Specification Description Language (SDL) [10].

As first step, L1 SW functional specification is created to capture system requirements in form of use cases and message sequence charts (MSC). Functional specification also deals with higher layer interfacing requirements and the programming requirements of L1 HW.

Detailed design specification is created as a next step and this acts as a basis for the implementation of L1 SW. This specification identifies different sub-systems to be implemented and refines the system scenarios by creating inter sub-system MSCs. Detailed design specification contains SDL block and state transition diagrams, message and interface specifications, abstract data types and procedure outline.

Finally, representative test scenarios are identified to create verification test plan. Test cases are described using MSCs that provide a convenient means to define stimuli to L1 SW (design under test) from higher layer and L1 HW (environment) and to show expected responses back from design under test to the environment. Also timing constraints are specified at this level.

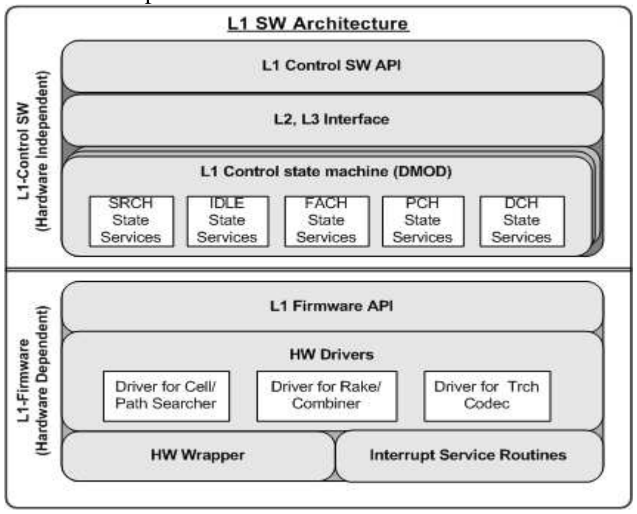

Figure3: SW architecture

The L1 SW performs tasks like idle mode and power save operation, measurement and data channel control, tracking and synchronization control, protocol data processing through the protocol stack API and configure appropriate registers in the HW. The implementation architecture is shown in Figure 3. The architecture is split into two parts: L1 control SW that is HW independent and L1 firmware that is tightly coupled with L1 HW.

L1 control SW implementation consists of:

- L1 control state machines (FSM), e.g. such as for demodulator control etc. Each FSM is mapped to an RTOS task that performs state dependent processing, for example, measurement control, physical channel configuration etc. Thus, each FSM contains several state service procedures. These procedures process events like primitives containing information from higher layers, intra-layer messages, interrupt service routine (ISR) notifications, timer events etc.
- The higher-layer interface to RRC (L3) and MAC (L2) supports primitive related functions. This interface takes care of centralized buffer management to avoid duplication of data that is used in other L1 SW modules. Furthermore, this provides modularity in the design such that variety of 3rd party protocol stack implementation can be plugged in by changing implementation of this interface alone without changing implementation of other modules in L1 SW. A control SW API is responsible for L1 software initialization and message passing.

L1 firmware handles HW dependent processing and performs the L1 HW configuration. It contains:

- Firmware API to provide API services to control state machine and to ISR in a state dependent manner
- Interrupt service routines that process periodic and intermittent interrupts from HW, dispatches message and events to control state machines, performs task and message prioritization, schedules and triggers firmware driver procedures for time critical functionality.
- HW drivers to implement firmware state machines triggered by calls from ISR or L1 control state machines, maintains module dependent HW/SW state information, configures HW and obtains state results from HW through HW wrapper.
- HW wrapper for accessing HW registers and to perform HW specific conversions.

L1 SW is implemented in ANSI-C for an ARM processor core using an RTOS. During and after implementation, suitable platform and methodology are required in order to satisfy verification needs. It is desirable that such platform facilitates verification at an early stage of software development cycle and also provides means to integrate abstractions for higher layer and HW, which will enable integrated verification (i.e. L1 HW, SW and protocol stack). The platform must also be capable of integrating the instruction set simulator on which L1 SW is running. Next section discusses the verification methodology in detail.

3. "Software-centric" Verification

In this section, we will describe the verification approach for performing the tests at various development stages.

Our aim is to use and reuse the software-specific verification models and test benches for all the steps. One main approach is to employ abstraction where appropriate. SystemC2.0 provides promising modeling paradigms. We will follow the definition of abstraction levels as described by the Open SystemC Initiative (OSCI)[11][12] and describe their relation to the verification steps in our case study.

Algorithm Level (AL): This level is commonly used for capturing the functionality. In our case data-flow models (floating-point, fixed-point) of the L1 HW has been developed based on ANSI-C and C++ within the CCSS environment as described in [8].

Programmer's View (PV): This level abstracts from particular bus architecture and makes use of a register map representation of the L1 HW. Accesses to HW registers are stubbed out and HW behavior isn't captured in a meaningful way. It offers up to 100 MHz execution performances, but when it comes to HW dependent processing and timed HW/SW interaction this approach comes with increased modeling overhead. Since in our case a lot of HW/SW interaction is required, we have conducted the L1 SW development at the PVT level described below.

Programmer's View + Timing (PVT): This level incorporates a timing-approximate view of the processor's peripherals and instruction set. In our case we make use of the ARMULATOR ISS (Instruction Set Simulator). Most commonly ISS operate at the PV-level taking only into account very basic instruction timing. These kind of ISS tools are suitable for initial functional design and verification, as well as for performing initial cycle-count estimation and profiling. This abstraction usually yields an execution performance in the order of 1 MHz.

Cycle Callable (CC): This level provides cycle-accurate modeling on the HW/SW interface based on a specific bus architecture model. At this level, the microprocessor architecture and its peripherals are modeled and this allows porting of RTOS to specific architectural variants. It abstracts from RT signals and provides execution performances up to 100 kHz range. Using CC-level processor simulators allows capturing the bus loads from instructions fetches as well as from accesses to HW peripherals. Thus this approach is used at the integration-level verification steps.

RT Level (RT): This level is used for the cycle-true representation of the L1 HW. For the SW verification steps it provides less applicability because of the slow execution times.

In the following sections, we will describe the general approach required for the separate steps in the SW verification flow as well as the seamless use and extension of the SW test environment.

3.1 Unit-level Testing

For unit level testing of different modules at the PVT-level, it is possible to reuse the simulation setup with no or little changes. For example, verification of demodulator control FSM functionality involves modeling of required interface abstractions for higher layer and HW. In order to implement test cases, concepts and methods of TTCN-3 [13] are used, and their semantics have been implemented on top of SystemC-based runtime library.

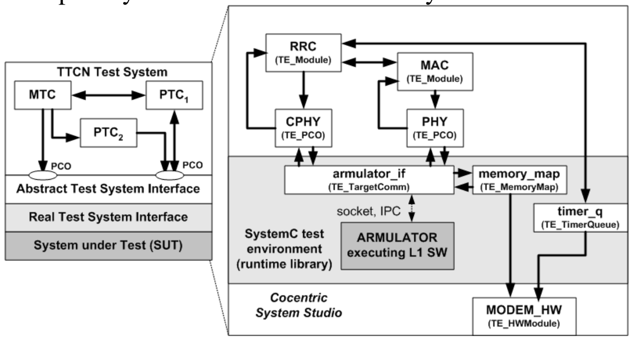

Figure4: Concept of TTCN-3 test configuration and mapping to SystemC Test Environment

Figure 4 shows the conceptual view of a TTCN-3 test configuration. Within every configuration there exist one main test component (MTC) and optional parallel test components (PTC). The MTC and PTCs implement particular test scenarios and communicate with the system under test (SUT) using dedicated points of control and communication (PCO). Communication between test components and with the PCOs in TTCN is FIFO-based.

On the right hand of Figure 4, it is shown how this basic concept is mapped to a SystemC configuration. All blocks are implemented using SystemC2.0. SystemC provides a base class sc_module, which is extended by a class TE_Module in order to add TTCN-specific capabilities and syntax. All interaction between blocks is mainly implemented using method call interfaces and event sensitivity providing high simulation speed. In the example, we implemented two test components for the protocol stack specific test behavior (Radio Resource Control - RRC, Medium Access Control - MAC), both inheriting from TE_Module. The two PCOs CPHY and PHY are in charge of transporting higher-layer primitive data from the RRC and MAC component in a FIFO manner. They are inherited from the transactor channel classes TE_PCO. To facilitate communication between SystemC test environment and the ISS, which is ARMULATOR in our case, interfaces are developed in both sides.

The ARMULATOR[14] offers a flexible and powerful API, which is customized for both, handling of register accesses to the L1 HW as well as for implementing a message based interface for SW/SW interfaces from higher protocol layers.

The class TE_TargetComm implements interfacing with ARMULATOR from SystemC side and provides all required communication to the L1 SW, by using the customized peripheral model of the ARMULATOR. This includes a serial interface for the higher-layer messages as well as a memory-mapped interface that redirects accesses to the L1 HW registers to the SystemC test environment.

Also the synchronization of simulation time, to emulate parallel discrete event simulation, is implemented using a time-stamped protocol. The ARMULATOR interface model dispatches every register access to the memory map model to which abstract HW models are attached. These models implement HW behavior and responses, such as interrupts and updating of registers. In addition register access tracing and monitoring functionality are provided in HW models. In order to specify time constraints and to model time-outs in the test components, a global timer queue model is provided in the runtime library, which in turn notifies the corresponding events once the timer has elapsed.

3.2 System-level Verification

For demonstrating system level verification, PLMN selection test case is described below as an example. When a UE is switched on, the first task is to select a public land mobile network (PLMN). Subsequently, UE searches for a suitable cell in the selected PLMN to camp on. At RRC level, this operation can be viewed as follows. The entire process is triggered by a measurement request from RRC to UE L1. After doing initial cell search operation, UE L1 returns a list of carrier frequencies and scrambling codes appended with cell-search information to RRC so that RRC can take decision on best cell.

At lower levels of operation, this involves series of firmware calls from L1 control SW for different operations such as frequency setting, configuring cell searcher HW, etc. Once the HW completes the search operation, it returns information by raising an interrupt to firmware. This gets packed into primitive and is returned back to RRC.

For verifying this system level task, the entire system test case is modeled on CCSS using SystemC. The Idle mode behavior of RRC is abstracted with a SystemC model. The interface between RRC and L1SW is abstracted as SystemC channel. L1SW is implemented on top of an RTOS and it is run on an ISS. The HW abstraction is modeled to take request from L1SW through systemC interface and to provide interrupts at appropriate times back to L1SW.

If the test-case has to be changed for some other scenario, for example for cell-reselection, it can be done by configuring appropriate datasets (or parameters) for RRC behavior model and HW models.

3.2.1 *Test case construction*

In this section, we illustrate how a test case is implemented using SystemC2.0 and what kind of test case specific constructs are used. In TTCN, each test component has a module control part. This is implemented as SC_THREAD (control) in the class TE_Module using a virtual function to be implemented by the child class. TTCN specifies a set of behavioral program statements targeted towards the specific requirements of test case construction. We implemented the semantics of a subset of these statements. Because TTCN is a notation and not an executable program, minor syntactical changes were required in order to map it to SystemC based solution.

```
void RRC::control() {
  int x=0;
  TE_Timer cs_timer (repitition_time_ms, SC_MS);
  TE_Message<CPHY_Measurement_Req>
                cphy_req(plmn_setup.str(),L23INT_PORT);
  cphy->send(cphy_req);
  cs_timer.start();
  TE_Message<CPHY_Measurement_Ind>
                cphy_ind(result_template.str());

  alt {
    alt_gbranch ((x<nbr_retries),cs_timer.timeout()) {
       setverdict(incon);
       cphy->send(cphy_req);
       cs_timer.start();
       x++;
       repeat();
    }
  alt_gbranch ((x==nbr_retries),cs_timer.timeout()) {
       setverdict(fail);
    }
  alt_branch (cphy->receive(cphy_ind) {
       setverdict(pass);
    }
  …
}
```

Figure5: Test case example

Figure 5 shows an example of the control thread implementation. In the control body, first a timer is defined and set to a specific repetition time. Next, a higher layer message is constructed from an input file to be sent to a specific SW message queue later on. Then it is sent to the CPHY PCO and the timer is started. The expected result back from the SUT is specified in another message definition. A template is provided for verifying the result.

A behavioral program statement in TTCN, an alt-statement, is also shown in Figure 5. An alt-statement denotes branching of test behavior, e.g. due to the reception and handling of communication and/or timer events, and it denotes a set of possible events that are to be matched against a particular snapshot. A similar semantic is implemented in the SystemC SW verification runtime library within the base class TE_Module from which each test component class is derived. In the example the condition of the alt-branches is checked at the moment a snapshot is taken. The first branch (guarded branch) is taken, if a local counter fulfills a certain condition and if a timer has elapsed. In this case a test verdict is set to inconclusive, another message is sent to the SUT via the PCO and the timer is restarted; the repeat-statement triggers a repetitive evaluation of an alt-statement. The last alt-branch is taken once a message has been received back, which matches the contents specified in the result template, and the test verdict is set to pass. Following this approach a test engineer is able to construct verification modules with a syntactical similarity to TTCN.

3.3 Integration-level Testing

The test environment as shown in Figure 4, is extended without much modification to perform verification of integrated L1 HW, SW and protocol stack. L1 HW in the test environment can be either cycle accurate HW models or abstract behavioral models or a mix of both. While using cycle accurate HW models at integration testing, it is also possible to model the actual bus interface e.g. using AMBA models with the ARM μP [5][15].

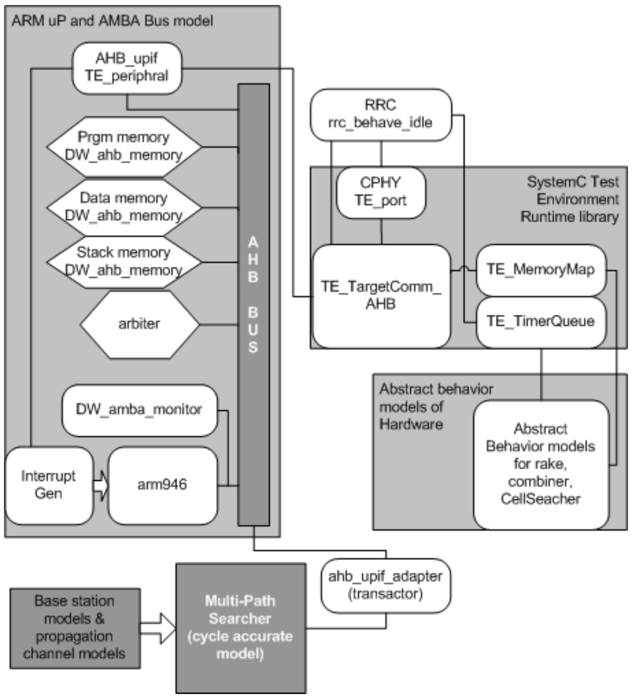
ARM uP and AMBA Bus model
AHB_upif TE_periphral
Prgm memory DW_ahb_memory
Data memory DW_ahb_memory
Stack memory DW_ahb_memory
arbiter
AHB BUS
DW_amba_monitor
Interrupt Gen
arm946
RRC rrc_behave_idle
CPHY TE_port
SystemC Test Environment Runtime library
TE_TargetComm_ AHB
TE_MemoryMap
TE_TimerQueue
Abstract behavior models of Hardware
Abstract Behavior models for rake, combiner, CellSeacher
ahb_upif_adapter (transactor)
Base station models & propagation channel models
Multi-Path Searcher (cycle accurate model)

Figure 6: "Software-Centric" SystemC-based Test Environment for Integration-level testing

Figure 6 shows an example of integration-level testing. Only one HW component (multi-path searcher) exists as cycle accurate model while the other HW components are represented as abstract behavioral models as required for this specific test scenario. Also, the µP bus interface is integrated in the test environment.

This verification flow can be extended to include FPGA based testing for board bring up once the L1 HW prototype is available. Concepts on how to integrate a FPGA prototype to a system-level simulation environment like CCSS are described in [16].

3.4 Interface Synthesis

The HW/SW interfaces, SW/SW interfaces to protocol stacks have to be included in the L1 SW code as well as in the SystemC test environment to establish communication between L1 SW and protocol stack as well as between L1 SW and L1 HW in the test environment. We have used XML-based approach to automate interface synthesis from the corresponding specifications. The automation ensures that derived interface specification is correct by construction and is consistent, considering the huge amount of interface data to be handled over SW-SW interface and several hundreds of registers that are to be accessed through SW-HW interface.

For the SW-SW interface we made use of the fact that in 3G standard, higher layer messages are based on Information Elements (IEs) and IEs are specified completely using abstract syntax notation or ASN.1 [17]. Higher layer primitives in 3G specification are generic and do not constrain implementation. We defined the implementation specific primitives in ASN.1 and they are based on IEs defined in the standard. These primitives along with defined IEs are given as input to a commercial ASN.1 to C compiler. The compiler generates the header files for L1 SW code and for the test environment.

HW-SW interface specification is driven by requirements of HW programmability, which in turn depends on 3G specifications itself and also on the implementation of HW controller. The starting point for HW/SW interface generation (design) is a table-based specification of register map that is an outcome of HW controller implementation. This specification is then converted using scripts to automatically generate:

- HW/SW interface in form of header file that is used for implementing HW wrapper part of L1 firmware.
- HW/SW interface in form of HDL that is used in HW µP interface implementation.
- C interface file that is used to model the µP interface in the SystemC based test environment.

4. Conclusion

In this paper, we described a complete design and verification flow for HW/SW implementation for a complex wireless IP. The versatility and reusability of the "software centric" verification flow was demonstrated using 3G L1 HW/SW as a case study, which can be applicable to other systems as well. The methodology presented includes a SystemC-based platform incorporating standardized Testing and Test Control Notation concepts for early verification of L1 SW at unit level, system level and for verification of integrated L1 HW, SW and protocol stack. There is a significant gain in terms of reuse of simulation environment to meet verification challenges at various levels of testing. An automated procedure for interface synthesis was presented, which ensured easy and consistent construction of huge amount of interface data.

5. References

[1] F. Longoni, et al, Radio Access Network Architecture, pp. 51-67, "WCDMA for UMTS", eds. H. Holma, A. Tosklala, Wiley, 2001.

[2] Jon Connell, Bruce Johnson , *Early Hardware/Software Integration Using SystemC 2.0*, ESC San Francisco 2002.

[3] J. Connell, ARM System-Level Modeling, Ver 1.0, June 25, 2003; http://www.arm.com/

[4] S. Pees, et al, *LISA Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures*, in Proc. of the DAC, June 1999.

[5] Synopsys ARM design ware (processor model), http://www.synopsys.com/products/designware

[6] Sudeep P,"Transaction Level Modelingof SoC with SystemC2.0", http://www.systemc.org/

[7] M.Caldari, et al,"SystemC Modeling of a Bluetooth Transceiver", DATE-2003

[8] H. Dawid, et al, "Efficient Design flow from System level to Hardware in CoCentric System Studio", in proceedings of the 2002 DATE, Designer's Forum, pp. 3-7, 2002.

[9] Unified Modeling Language UML, Rational , http://www.rational.com/uml/resources/documentation

[10] ITU, *Z.100: SDL formal definition, Introduction*

[11] Th. Grötker, et al, System Design with SystemC, Kluwer Academic Publishers Group, 2002. ISBN.

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

[13] ETSI, *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 core language*, ETSI ES 201 873-1, August 2002.

[14] Armulator, ARM, http://www.arm.com/devtools/

[15] Synopsys AMBA design ware (verification IP), http://www.synopsys.com/products/designware/dwverificationlibrary.html - amba

[16] A. Müller, G. Post, and M. Vaupel, *RAVEN – A Real-time Analysis and Verification Environment*. In Proc. of the Int. Conference on Signal Processing Application and Technology (ICSPAT), pages 297-301, Toronto, Canada, Sept. 1998.

[17] Abstract Syntax Notation One (ASN.1) standard, ITU Rec. X.680-X.693, http://www.itu.int/ITU-T/asn1