EURASIP Journal on Embedded Systems

C-Based Design of Heterogeneous Embedded Systems

Guest Editors: Christoph Grimm, Axel Jantsch, Eugenio Villar, and Sandeep K. Shukla

C-Based Design of Heterogeneous Embedded Systems

EURASIP Journal on Embedded Systems

C-Based Design of Heterogeneous Embedded Systems

Guest Editors: Christoph Grimm, Axel Jantsch, Eugenio Villar, and Sandeep K. Shukla

Contents

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 243890, 2 pages
doi:10.1155/2008/243890

Editorial
C-Based Design of Heterogeneous Embedded Systems

Christoph Grimm,[1] Axel Jantsch,[2] Sandeep Shukla,[3] and Eugenio Villar[4]

[1] *Vienna University of Technology, Vienna 1040, Australia*
[2] *Royal Institute of Technology, 100 44 Stockholm, Sweden*
[3] *Virginia Tech, Blacksburg, VA 24061, USA*
[4] *University Cantabria, 39005 Santander, Spain*

Correspondence should be addressed to Christoph Grimm, grimm@ict.tuwien.ac.at

Received 14 July 2008; Accepted 14 July 2008

With the proliferation of all kinds of electronic devices and our increased dependences on electronics in every aspect of our lives from handheld PDAs, cell phones, medical and prosthetic devices to automotives and fly-by-wire airplanes, embedded systems have changed dramatically. Such changes have given rise to increasingly complex applications on resource constrained embedded platforms, and new innovative system architectures. On the application side, embedded systems are not any more only invisible computers that do one dedicated task, but also ones which sense, observe, decide, act, and are networked with their environment (ambient intelligence).

Obviously, the extended role of embedded systems is not without impact on the hardware and system architecture. Embedded architectures have evolved from a low-end processor with application-specific coprocessors to networked multiprocessor systems including sensors, analog/mixed-signal, and RF components, giving rise to a high degree of heterogeneity which in turn requires sophisticated and heterogeneous modeling techniques. Because these systems are within the space of the physical world requiring interaction with analog physical phenomena, and communicate through radio-frequency (RF) communication links, the newer architectures are aptly referred to as "embedded analog/mixed-signal systems" (E-AMSs). This nomenclature captures the new complexity and heterogeneity that arise from the fact that HW/SW systems and analog, mixed-signal, RF blocks are so functionally interwoven.

To handle this growing complexity and heterogeneity, new methods and tools are required that are able to handle such intricate and closely coupled heterogeneity between software, hardware, and other (e.g., analog/RF) components. Even though the standard hardware description languages such as Verilog and VHDL have been extended over the years to model analog and mixed signal designs, the complexity ensuing from the close interaction of hardware, software, and mixed signal domains, and the multiple models of computation has necessitated newer system level languages and surrounding methodologies. One pragmatic approach championed by researchers and the industry has been to use existing software languages such as C and C++ for the design of the overall systems due to the dominant role of software. This helps tremendously in hardware/software cosimulation in the early stages of design, and also allows one to dynamically experiment with hardware/software partitioning for better performance. Among these C-based system level languages, SystemC has evolved as an industry standard for design of HW/SW systems—but with limited ability to deal with complex issues that arise when designing E-AMS systems. Hence, more research and development of tools and standards are required in this field.

This special issue on "C-based design of embedded systems" of the EURASIP journal on Embedded Systems includes 10 articles that will certainly give the readers an overview on the ongoing research towards a design technology that integrates all kind of components for the realization of future "ambient intelligence" systems.

The first two articles offer a deeper insight into the problems and also potential solutions: the first paper "Power aware simulation framework for wireless sensor networks and nodes" by Glaser et al. deals with new methods to optimize power consumption in energy self-sufficient sensor networks, based on SystemC and OMNET. The second paper "Modeling field bus communications in mixed signal

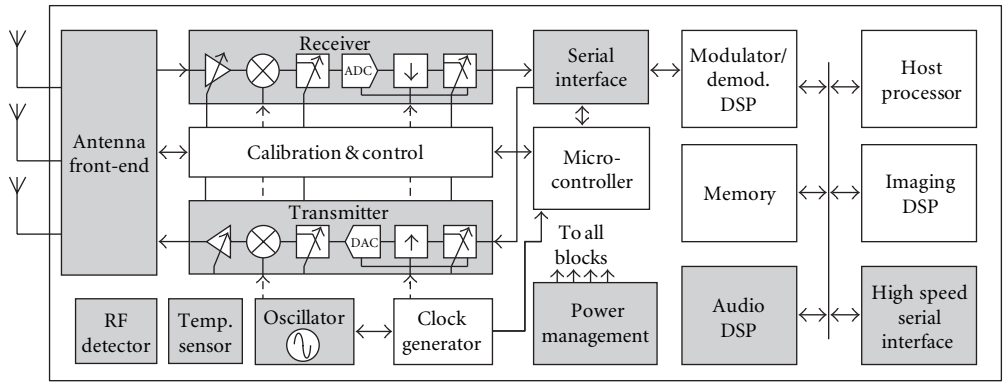


Figure 1: Example of an embedded analog/mixed-signal architecture: communication system (from: "*An Introduction to Modeling Embedded Analog/Mixed-Signal Systems using SystemC AMS Extensions*"; http://www.systemc.org/).

embedded systems" by Denoulet et al. uses the upcoming OSCI SystemC AMS extensions to deal with heterogeneity. Although AMS extensions still are changing, this case study also gives a valuable insight into capabilities of this upcoming standard.

A second group of contributions focuses on new tools and languages. The third paper "System-on-chip environment: A SpecC-based framework for heterogeneous MPSoC design" by Doemer et al. describes a design framework and methodology for heterogeneous MPSoCs based on SpecC, while the fourth paper "Novel methodology for functional modeling and simulation of wireless embedded systems" by Morales et al. addresses modeling and simulation of wireless embedded systems by bringing together the functional models of the baseband algorithms written in C language with the circuit descriptions at behavioral level in Verilog or Verilog-AMS in a single kernel environment, most notably from the viewpoint of the (semiconductor) industry. The fifth paper "Bridging MoCs in SystemC specifications of heterogeneous systems" by Haase et al., also in the context of the upcoming SystemC AMS extensions, describes some means to support the design refinement process targeted by the AMS extensions.

At a higher level, the abstract modeling, IP integration, and synthesis are of crucial importance which are tackled by the sixth paper entitled "Combining UML2 application and SystemC platform modeling for performance evaluation of real-time embedded systems" by Kreku et al., and the seventh paper "Automated integration of dedicated hardwired IP cores in heterogeneous MPSoCs designed with ESPAM" by Nikolov et al., dealing with automated integration of IP cores in heterogeneous MPSoCs. High-level synthesis and its impact on the design of heterogeneous embedded systems are evaluated in the eighth article with the title "Quantitative evaluation of the impact of high-level synthesis on heterogeneous embedded systems design" by Hammami et al.

Last but not least: validation. Validation of E-AMS is may be the most challenging issue. Approaches to improve verification coverage range from model-driven validation in the ninth article "*Model-driven validation of SystemC designs*" by Patel et al. to simulators that allow designers to integrate accurate behavior of analog and RF behavior into an overall system simulation in the tenth contribution with the title "*System level modeling of RF IC in SystemC-WMS*" by Orcioni et al.

We sincerely hope that this special issue will provide the readers with a good understanding of ongoing research and a sense of the challenges on the road ahead.

Christoph Grimm
Axel Jantsch
Sandeep Shukla
Eugenio Villar

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 369178, 16 pages
doi:10.1155/2008/369178

Research Article

Power Aware Simulation Framework for Wireless Sensor Networks and Nodes

Johann Glaser, Daniel Weber, Sajjad A. Madani, and Stefan Mahlknecht

Institute of Computer Technology, Technical University of Vienna, 1040 Wien, Austria

Correspondence should be addressed to Johann Glaser, glaser@ict.tuwien.ac.at

Received 1 October 2007; Revised 21 February 2008; Accepted 16 May 2008

Recommended by Sandeep Shukla

The constrained resources of sensor nodes limit analytical techniques and cost-time factors limit test beds to study wireless sensor networks (WSNs). Consequently, simulation becomes an essential tool to evaluate such systems.We present the power aware wireless sensors (PAWiS) simulation framework that supports design and simulation of wireless sensor networks and nodes. The framework emphasizes power consumption capturing and hence the identification of inefficiencies in various hardware and software modules of the systems. These modules include all layers of the communication system, the targeted class of application itself, the power supply and energy management, the central processing unit (CPU), and the sensor-actuator interface. The modular design makes it possible to simulate heterogeneous systems. PAWiS is an OMNeT++ based discrete event simulator written in C++. It captures the node internals (modules) as well as the node surroundings (network, environment) and provides specific features critical to WSNs like capturing power consumption at various levels of granularity, support for mobility, and environmental dynamics as well as the simulation of timing effects. A module library with standardized interfaces and a power analysis tool have been developed to support the design and analysis of simulation models. The performance of the PAWiS simulator is comparable with other simulation environments.

1. INTRODUCTION

The advances in distributed computing and micro-electro-mechanical systems (MEMSs) have fueled the development of smart environments powered by wireless sensor networks (WSNs). WSNs face challenges like limited energy, memory, and processing power and require detailed study before deploying them in the real world. Analytical techniques, simulations, and test beds can be used to study WSNs. Though analytical modeling provides a quick insight to study WSNs, it fails to give realistic results because of WSN-specific constraints like limited energy and the sheer number of sensor nodes. Real world implementations and test beds are the most accurate method to verify the concepts but are restricted by costs, effort, and time factors. Simulations provide a good approximation to verify different schemes and applications developed for WSNs at low cost and in less time. The available simulation frameworks are either general purpose or WSN specific. The general purpose network simulators do not address WSN specific unique characteristics while WSN specific simulators mostly lack the capability of capturing and analyzing the power consumption and timing issues at the desired level of granularity.

The proposed PAWiS simulation framework [1, 2] assists in developing, modeling, simulating, and optimizing WSN nodes and networking protocols. It particularly supports detailed power reporting and modeling of wireless environments. A typical WSN node may comprise various types of sensors (e.g., temperature, humidity, strain gage, pressure), a central processing unit (CPU) with peripherals, and a radio transceiver. The simulation covers the internal structure of these nodes as well as communication among them. Sensor nodes forming a network communicate with each other via an ad hoc multihop network. The range of applications that can be simulated covers many domains such as building automation, car-interior devices, car-to-car communication systems, container monitoring and tracking, and environmental surveillance.

The PAWiS simulation framework provides a way to reduce the overall power consumption by carefully optimizing various design aspects within the context of the application. Enhancing energy performance could propel

many new applications since the lack of sufficient battery lifetime or limited energy scavenging systems is still the main cause for the slowly spreading number of WSN applications.

In previous research performed at the Vienna University of Technology [3], several weaknesses of current WSN nodes were identified. These include the wakeup problem (i.e., how to wakeup a sleeping node), the voltage matching and power supply problem, the fairly long oscillator start-up time, and other hardware related problems. However, overall efficiency also strongly depends upon the application and its interaction with other nodes and the environment. Here, communication protocols play an important role, but considering the different layers of protocols independently and not taking into account adjacent layers as well as the hardware and environment, improvements can only be suboptimal [4]. PAWiS explicitly supports cross-layer design to exploit the synergy between layers. Several aspects regarding power aware wireless sensors are emphasized and directly supported by the PAWiS framework. The PAWiS framework hence helps to capture the whole system in one simulation and extracts power consumption figures from software and hardware modules uncovering leakages in early design stages. The main contributions of this work are as follows.

(i) One of the main contributions is to equip the user to program models of a wide variety of abstraction,

(ii) Another contribution is to model the internals of WSN nodes as well as the communication between them. The framework distinguishes between software and hardware tasks, yet it is easy to change the hardware/software partitioning,

(iii) one of the main contributions also is an elaborate power simulation with any level of accuracy which can still be balanced with complexity. The simulated power consumption can depend on the supply voltage, for example, for a nearly empty battery when supplying a microcontroller that operates at very low voltages. In WSN nodes, components with different supply voltages are combined resulting in the need for low dropout regulators (LDOs) and DC/DC converters. The PAWiS framework allows to model this hierarchical supply structure as well as the efficiency factor of the converters.

(iv) Powerful analysis and visualization techniques are provided to evaluate the simulation results and derive a path to optimization.

(v) The RF communication is modeled according to real-world wave propagation phenomena while still maintaining an efficient simulation. It includes interferers, noise, and attenuation due to distance to influence the bit error ratio of communication links. No preset topology is required because the packets are transmitted to all nodes within reach. The topology of network communication itself originates from the link quality and the routing algorithm. With this approach any routing protocol, especially ad hoc protocols, can be implemented. The transmission model implementation is entirely independent of the underlying modulation format enabling the simulation of any type of modulation. Multiple participants can utilize the RF channel by multiple access schemes and are separated by space, time, frequency, and code.

2. RELATED WORK

To have credible results through simulation, the choice of models and the simulation environment is very important. Key properties for WSN simulators must include a way to capture energy consumption at any level of abstraction, powerful scripting language, graphical user interface (GUI) support to animate, trace, and debug, and the ease to integrate new modules. Some of these key properties are discussed in [5].

NS-2 is a discrete event, object-oriented, general purpose network simulator written in C++ (http://www.isi.edu/nsnam/ns/). According to [6], it is the most widely used simulator and has a rich library of protocols but focuses mainly on IP networks. OTcl [7] is used as scripting language to control and configure simulations. It provides a GUI support with the network animator (Nam) which is not so good and only reproduces NS-2 trace [5]. For WSN, NS-2 does not scale well, and it is difficult to simulate 100+ nodes [8]. NS-2 lacks detailed support to measure the energy utilization of different hardware, software, and firmware components of a WSN node. "*One of the problems of ns2 is its object-oriented design that introduces much unnecessary interdependence between modules. Such interdependence sometimes makes the addition of new protocol models extremely difficult, which can only be mastered by those who have intimate familiarity with the simulator*" [9].

SensorSim [10] is an NS-2-based simulator for modeling sensor networks. The authors have provided a power model, a battery model, and a CPU model to address sensor network specific constraints but because of the "*unfinished nature of the software,*" the simulator is no longer available (http://www.nesl.ee.ucla.edu/projects/sensorsim/).

OMNeT++ [11] is a discrete event, component based, general purpose, public source, modular simulation framework written in C++ (http://www.omnetpp.org/). It provides a strong GUI support for animation and debugging. The mobility framework (MF) [12] for OMNeT++ is a specific purpose addon to simulate ad-hoc networks. In the MF, the links between pairs of neighbor nodes are specified with OMNeT++ gates with the help of an additional global module called channel control. Unlike "*visible communication paths*" which are created and freed in bulk, the air module of the PAWiS framework (see Section 3.3.7) decides about connection between pairs of nodes dynamically based solely on their position information and other radio transmission effects. Another difference is the capturing of power consumption with the power meter (see Section 3.3.8) that helps to identify major energy consuming modules. OMNeT++/MF does not provide a WSN specific module library [13] which can help expedite the design process.

SenSim [14] is an OMNeT++ based simulation framework for WSN. Some protocol layers and hardware units are implemented as simple OMNeT++ modules. They have provided implementations for different WSN specific protocols, battery, a simple CPU implementation, a simple radio, and a wireless channel. A coordinator module is implemented to assist in inter-communication between hardware and software modules. It does not add additional functionality to OMNeT++ other than a simulation template with a small number implemented modules.

NesCT (http://www.nesct.sourceforge.net/) is an add-on for OMNeT++ which allows the simulation of TinyOS-based sensor networks in OMNeT++ (http://www.tinyos.net/). It does not come with any additional functionality but acts as a language translator between two different environments (TinyOS and OMNeT++).

Global mobile information system simulator (GlomoSim) [15] is a library-based general purpose, parallel simulator for wired and wireless networks written in Parsec (http://www.pcl.cs.ucla.edu/projects/parsec/) (C-based discrete event simulation language for parallel programming). Being parallel, it is highly scalable and can simulate up to 10 000 nodes [5]. Using GlomoSim requires learning the new language Parsec. GlomoSim is superseded by QualNet (http://www.scalable-networks.com), a commercial network simulator and is not released with updated versions since 2000. However, sQualNet [16], an evaluation framework for sensor networks, built on the top of QualNet has been released recently.

OPNet Modeler is a commercial, well-established (1986), general purpose, object oriented simulation environment written in C++ (http://www.opnet.com). It supports discrete event, hybrid, and analytical simulation. It provides a very rich set of modules for all layers of protocol stacks including the IEEE 802.11 family, IEEE 802.15.4, and routing protocols like AODV [17], and DSR [18]. Each level of the protocol stack has to be implemented as a state machine but it is *"difficult to abstract such a state machine from a pseudo-coded algorithm"* [19]. The authors in [19] compared OPNet Modeler, GlomoSim, and NS-2 with a broadcast protocol. The results show that the performance of NS-2 and GlomoSim, and OPNet is barely comparable.

SENSE [9] is a sensor network specific, component-based simulator written in C++ built on the top of COST [20]. To address the issue of scalability, SENSE provides an optional way for parallel simulation as is done in GlomoSim. It provides a small library of module implementations like AODV and DSR, with simplistic battery and power models; but unlike PAWiS, it does not provide a detailed structure to capture energy consumption of different hardware and software components. It also lacks a visualization tool which is helpful in debugging and visual inspection.

Ptolemy II [21] is a component assembly-based software package to study concurrent, real time, and heterogeneous embedded systems and is written in Java. Ptolemy II provides a rich support to model, simulate, and design components in different domains (e.g., discrete time or component interaction), but according to recent Ptolemy II 7.0.beta release notes, its wireless domain is still in experimental phase (http://www.ptolemy.berkeley.edu/ptolemyII/). VisualSense [22] is an open source WSN visual simulation framework built on Ptolemy II.

J-Sim [23] is a general purpose, component-based, open-source simulation framework written in Java. It is glued to different scripting languages with TCL/Java and hence is a dual language framework like NS-2. Initially, designed for wired networks, its WSN specific package provision supports only 802.11 MAC scheme and some high-level models, for example, battery, CPU, wireless channel, and sensor channel.

Various WSN specific simulation and emulation tools have been released in the previous years. These tools include TOSSIM [24], EmStar [25], and ATEMU [26]. The advantage of using such tools is that the code that is used for simulation/emulation also runs on the real node (with minor modifications) reducing the effort to rewrite the code for the sensor node and giving detailed information about resource utilization. The main problem with such frameworks is that *"the user is tied to a single platform either software or hardware (typically MICA motes), and to a single programming language (typically TinyOS/NesC)"* [5]. Tython [27] and PowerTossim [28] are extensions of Tossim to capture the dynamic behavior of environment and power consumption, respectively.

In contrast to many of the above frameworks, the PAWiS simulation framework meets all the key properties outlined in [5]. It also utilizes the powerful GUI support of OMNeT++, it utilizes the widely used scripting language Lua to include environmental dynamics and mobility, and to reduce the test-debug cycle (http://www.lua.org/). It focuses on capturing energy consumption at hardware and software levels, provides a visualization tool to analyze energy consumption, a rich library of modules to get an optimized protocol stack, standardized interfaces to improve reusability, and provides a simulation template for the user to jump start any simulation study.

3. SIMULATION FRAMEWORK

A wireless sensor network is built of independent nodes which communicate via an ad hoc wireless network. The PAWiS simulation framework is designed to model, simulate, and optimize both the wireless communication protocols as well as the interior of the nodes. Each node is built as a virtual prototype in a way that its function, timing, and power consumption as well as system failures are simulated at any level of detailedness.

3.1. Methodology

The design of a WSN and its nodes follows a top-down approach embedded in a cyclic process. The functional specification defines requirements of the WSN which apply to the architecture as well as to the implementation. The implementation on the other hand imposes constraints on the architecture and the functional specifications in a bottom-up manner. For example, the functional specification of a tire pressure monitoring system (TPMS, see also Section 6.2) may require the sensors to measure the current pressure

every 20 seconds. Due to the power consumption of the current sensor technology and the available battery capacity, the implementation of a TPMS sensor node imposes the constraint that this function can only be maintained for 2 months, which is rather short compared to the lifetime of a tire.

3.1.1. Work flow

To design a WSN and its nodes, the functional specification is defined. For a full optimization, the work flow is a cyclic application of the following steps.

(a) Every node typically consists of multiple submodules. In this step, the node structure is defined and for every module type a certain implementation is chosen (*composition*). Initially, the modules only need to meet minimum functional requirements. For instance, for the aforementioned TPMS node, the user chooses a CPU, a pressure sensor, an AD converter, a timer, an RF transmitter, and memory. Additionally, software modules like the network stack and sensor handling are required.

(b) The modules chosen in the previous step are *integrated*. Their interfaces have to be adopted and their functions must be coordinated. For example, the chosen pressure sensor might require special treatment for its power-on sequence, which must be implemented accordingly in the module which operates it.

(c) The modules are *configured*, that include setting values for the clock frequency of the CPU, the resolution of analog-to-digital converters (ADCs), and so forth.

(d) In the previous steps, a fully functional model of a node and of the network was set up and is *simulated* in the current step.

(e) The simulation results are *evaluated*. This includes the verification of the function, analysis of the power consumption and timing, and detection of potential for further optimization (see Section 5).

(f) The issues identified in the previous step are considered for a *refinement* of the models as well as the design. Examples are increasing the detailedness and accuracy of power consumption and timing, dividing the functionality into more elaborate modules, configuration changes, and even a modification of the node composition by exchanging module implementations. The chosen pressure sensor might consume too much energy in every measure cycle due to its long-lasting power-up sequence, and hence should be exchanged by a sensor with faster startup. Another example is the physical layer model (including the RF transceiver) which might need a refinement of the power consumption reporting during intermediate states (e.g., when switching from transmit to receive mode) (see Section 3.3.8). With the refined node implementation, the procedure is started over again, until the optimization goal is achieved.

These *refinement cycles* are the main track to enhance the development and design [3]. After completing the optimization process, the final outcome comprises the verified function, the architecture of the node, the implementation details, and the power specification of every module.

The module library (see Section 4) is particularly intended for the composition and integration of modules to a node. It provides a collection of multiple module implementations for every module type which can be combined in numerous ways. The integration effort is minimized because these modules conform to the interface specification (Section 4.2).

3.1.2. Optimization

Several strategies for the optimization of WSN nodes are proposed in [3]. The PAWiS simulation framework is especially constructed to assist the designer in applying these strategies.

(i) *System-level optimization* involves a modification of the whole system behavior like choosing a different network layout or application patterns.

(ii) Exchanging the *module implementation is done* by selecting a different module from the library. For example, choose a dual-slope, a $\Sigma\Delta$ or an successive approximation ADC. Another example is to change a communication layer implementation (e.g., use another medium access (MAC) protocol).

(iii) Another strategy is exchanging *multiple module implementations* for adjacent modules that tightly work together. While modifying a single module might degrade the node performance, the interaction of the changes of multiple modules potentially leads to an overall improvement.

(iv) *Cross-layer optimization* works on more than one network layer where, for example, modifying the routing protocol benefits from a different physical layer.

(v) *Partitioning* of modules and/or functions is done by dividing the task between hardware and software, digital and analog, or RF and baseband. For example, a specific MAC protocol could be implemented in software, as dedicated hardware acceleration unit, or a combination of both.

(vi) *Another strategy is scaling* a module, for example, the resolution of an ADC or the register count of a CPU.

(vii) *Parameterization* of modules, for example, the timing, transmission power, and bit rate of a radio transceiver.

3.2. Structure

The PAWiS simulation framework is based on the OMNeT++ discrete event simulator [11] which is written in the C++ programming language (Figure 1). A discrete event simulation system operates on the basis of chronological

consecutive events to change a system's state. These events are processed by the simulation kernel. The simulation time itself does not progress continuously but is advanced with each occurring event (hence it is not possible to issue events that are scheduled before the current simulation time). The proposed framework handles timing-related issues according to this discrete event mechanism.

User-defined models are implemented as C++ classes and mostly utilize framework concepts. The user of the framework is only confronted with OMNeT++ to comprehend the simulation process. Node composition and network layout along with environmental and setup parameters are specified in configuration files as well as script files (see Section 3.4). The modules are compiled and linked with the simulation kernel, and result in the simulation application. This offers a GUI-based frontend which enables visual debugging of the communication processes of the model on a per-event basis at simulation runtime. An optional command line-based frontend can be utilized for increased simulation performance.

The framework is primarily focused on simulating inter- and intranode communication. Additionally, fine-grained aspects (e.g., CPU instruction set emulation as used by [26]) can be easily modeled with user extensions. However, a tradeoff between simulation details and execution performance (as discussed in [29]) has to be considered with increasing quantity of network nodes.

A promising feature is the possibility to use SystemC in combination with OMNeT++ (http://www.systemc.org/). This is achieved by combining the OMNeT++ simulation kernel and the SystemC simulation kernel in a way that events from OMNeT++ and SystemC are being processed together. This also allows the communication between both domains. OMNeT++ allows the use of a custom scheduler which is the central point to merge the messages and events of OMNeT++ and SystemC, respectively. Unfortunately, the OSCI SystemC kernel does not offer such an interface, so slight modifications in the source code were necessary.

Simulation results comprise timing and power consumption profiles as well as event records. The completed model itself contains information regarding the functional description and architecture specifications along with low-level implementation details.

3.3. Basic concepts

3.3.1. Modularization

A wireless sensor node is typically composed of multiple *modules* (e.g., CPU, timer, radio, network layers). Internally, every module is based on one or more *tasks*. The framework defines two types of tasks. One type models a *hardware component* (e.g., a timer, an ADC) whereas the second type is a *software task*, for example, application, routing, MAC, and physical layer. Every module is implemented as a C++ class derived from a framework base class. Tasks are implemented as methods within a module class. The execution of a single task is sequential but all active tasks are running in parallel. This form of concurrency is implemented as cooperative multithreading, where the program flow is suspended within a method when certain framework calls are made (e.g., to wait for some condition to be satisfied) and will continue execution after being dispatched again. This process is transparent for the user and is entirely handled by the framework.

Basically, the detailedness and granularity of the sensor node model strongly depend on the design and simulation requirements for hardware and software modules and are not restricted by the PAWiS framework.

3.3.2. Functional interfaces

Control flow transitions between two modules are specified by the so-called *functional interfaces* (FI). They can be thought of as subroutines with well known names and parameter specifications. An invocation of an FI is similar to a blocking subroutine call but may exceed the module boundary. The framework allows the passing of arguments to and from FIs. In the model, FIs are implemented as class methods (similar to tasks).

A collection of FIs grouped together under a well-known name can be thought of as a functional module-type description. This introduces a level of abstraction in the functional design process and hence enables reusability of functional design. Two modules that are completely satisfying the specification regarding their FIs can be said to be functionally equivalent (although they might have entirely different power consumption and timing profiles). This approach is utilized in the module library (see Section 4).

3.3.3. CPU

As already mentioned in Section 3.3.1 tasks can either model hardware or software. Software tasks of sensor nodes are executed by a CPU. It is important to note that multiple software tasks cannot run in parallel, since typically only one CPU is available and supported by the framework. The CPU module of the framework ensures that only one task's code simulation is executed at a time.

To model the power consumption and timing behavior of software tasks, the PAWiS simulation framework splits the simulation into two parts. The functional part is implemented in the C++ method of the task. The timing and power consumption part, on the other hand, is delegated to the CPU module which maintains its power consumption and delays execution of the software task for the calculated processing time (the time that the code execution on the CPU would take). This means that the whole functional part is executed at the very same simulation time instant. The model programmer has to insert special framework requests to the CPU module to simulate the execution time and power consumption.

These requests include the estimated execution time of the firmware code on the CPU. Now think that the CPU of a given node should be replaced during the optimization process. This would also require to modify all execution time estimates in all modules of the node. To allow for a CPU exchange without the need to adapt other modules the

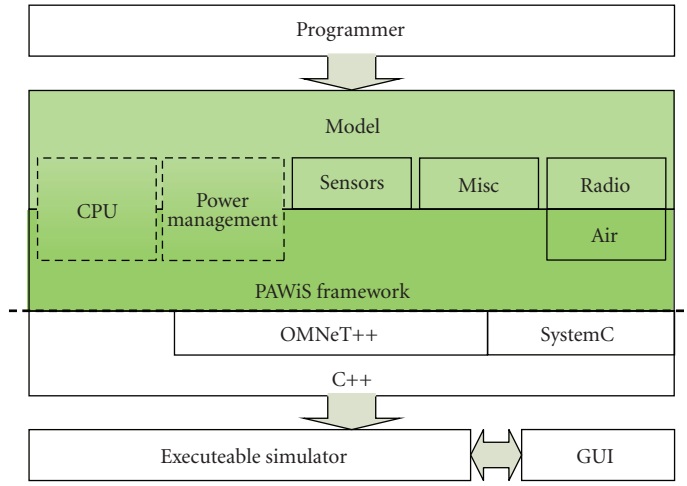


Figure 1: Structure of the PAWiS simulation framework.

execution time estimates are referred to the so-called *norm CPU*. This is an imaginary but well-defined CPU implementation (regarding its performance). The actual CPU model scales its processing time and power consumption according to its individual properties. For higher accuracy, the CPU request also supplies the percentage of integer, floating point, memory access, and flow control operations.

Many microcontrollers used for WSN nodes have CPUs which offer special low-power modes. The PAWiS simulation framework also supports modeling of these states. This is done by pausing code execution and setting the power consumption of the CPU module to a lower value. To exit the low-power mode an interrupt can be issued.

3.3.4. Timing

Modeling time delays differ whether they occur in firmware or hardware modules. For hardware modules the framework provides a simple wait method to suspend execution for a certain amount of time. Several distinct implementations of the wait method are available with support for fixed and conditional timeouts.

Using wait is not valid for software tasks because it is not possible to wait and do nothing in software (even for an infinite loop without body, the CPU does something). In fact, if delays are needed in software, the corresponding module has to use a loop (or a similar construct) to wait for a certain time and therefore utilize the CPU to achieve the delay. The framework offers a variety of methods to utilize the CPU for timing and flow control purposes. Alternatively the CPU can be put to a low-power mode which stops execution too and therefore delays until an external or timer event occurs.

An important consequence of this timing model is that consecutive user code lines without a wait call or a CPU utilization request take place in the same simulation time instant (i.e., no simulation time elapses during that code execution). Simulation time only advances when these special methods are invoked.

3.3.5. Interrupts

The framework provides a basic mechanism to model interrupt handling in a two-step process that maps

(i) interrupt sources to interrupt vectors; and

(ii) interrupt vectors to interrupt service routines.

Whenever an interrupt request is issued, the framework handles the necessary task scheduling according to the interrupt priorities and the currently running task.

The implemented interrupt model supports several user configurable interrupt sources (potentially coming from different modules, e.g., a timer, an analog-digital-converter, etc.). Each of these sources is mapped to an interrupt vector. Additionally, it is possible to map multiple sources to one specific interrupt vector. Furthermore, every vector maintains a priority and an interrupt service routine (ISR). As the model allows multiple vectors to share one ISR, the framework provides means to identify the triggering vector from the ISR. The framework's CPU module entirely handles the interrupt processing except for prioritizing of interrupt vectors that needs to be provided in the user model by overriding the CPU base class.

The user can register ISRs for interrupt vectors within software modules. When interrupt sources trigger interrupt requests, the CPU module determines the appropriate interrupt vector, checks its priority, and if appropriate transfers control to the ISR (which is always a software task). In case of a control transfer, the currently executed CPU task is preempted and continues execution after the ISR has finished.

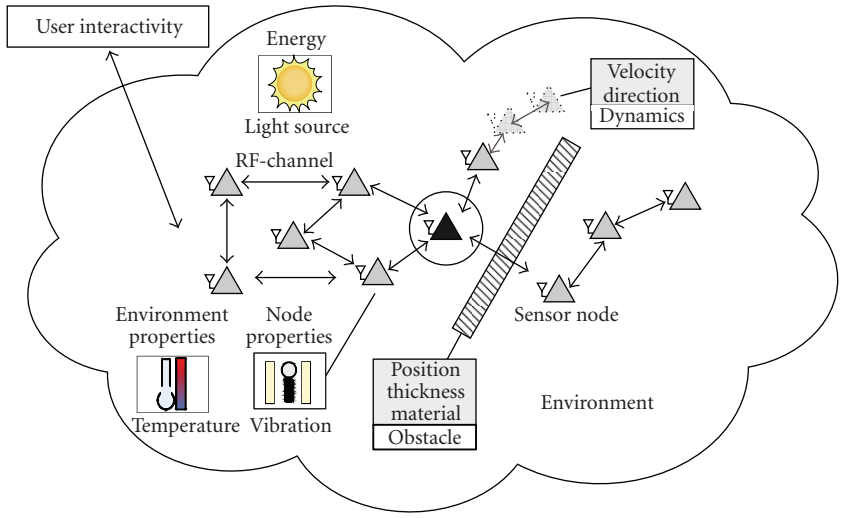


Figure 2: The Environment with properties, objects, and sensor nodes.

3.3.6. Environment

All sensor nodes are placed at 3D positions within the *environment*. This is a representation of the outer world and surroundings of all nodes including the RF channel.

Besides the nodes themselves, additional objects like walls, floors, trees, interferers, heaters, light sources, global properties (e.g., the attenuation exponent b (see Section 3.3.7)), and more are defined within the environment. The entire *environment* can be configured with configuration and scripting files.

3.3.7. Air

The *air* is an essential part of the *environment* to handle the RF channels, which are defined by 3D node placement in space and obstacles between the nodes. A real RF signal is subject to wave propagation phenomenons like attenuation, reflection, refraction, and fading (multipath propagation) from the transmitter to the receiver. In the PAWiS simulation framework, these effects can be modeled but currently we use a distance-based path loss radio model, which only considers the distance between transmitter and receiver.

The packet transmission is modeled without the definition of a predefined topology (similar to a wired network). Instead of that, every RF message is transmitted to all other nodes and the received RF power is calculated from the transmitter power, antenna properties and especially the distance and obstacles between the transmitter and the receiver. The topology of the network results from the reachability between nodes which is limited by the minimum received signal quality.

Signal power

The received signal power of a node is proportional to the transmitter power, only scaled by wave propagation effects and node properties. These constant attenuation factors between all nodes can be conflated to a matrix which is referred to as *adjacency matrix* within the framework. A simple row multiplication is used to calculate the received signal power for all nodes. The matrix is a precisely defined interface from the *Environment* setup (i.e., node positions, obstacles) to the data communication. Therefore, it can also be calculated by an external RF channel simulation tool.

The current implementation of the *Air* supports only isotropic antennas with uniform antenna gain. Obstacles are considered for the adjacency matrix by explicitly given additional attenuation factors between pairs of nodes in the *Environment* configuration.

Packet transmission

Whenever a data packet is transmitted by a node, the *Air* calculates the received signal power for all other nodes and notifies every node (above a certain threshold) about the start of the transmission. The nodes confirm the acceptance of the data packet, if their receiver is currently in listen mode and if the signal power is above the sensitivity threshold. During the transmission, the receiving nodes calculate the signal-to-noise ratio (SNR) between the received signal power P_{signal} and the received and internal noise power P_{noise} : $\text{SNR} = P_{\text{signal}}/P_{\text{noise}}$.

From this SNR, the bit error ratio (BER) is calculated, which is a function of the SNR depending on the (fixed) modulation format (the formula can be provided by the user). From the BER and the bit length of the transmission, the bit error count is calculated. Consequently, the user's module decides whether the received packet is valid or treats it as corrupted.

From the size of the transmitted data packet and the bit rate, the Air calculates the duration for the transmission. At the time when the transmission is finished, the Air notifies all receiving nodes again. This notification contains the user data, its length, and the number of bit errors. Additionally, as some protocols decide whether to process an incoming

packet after the arrival of some header fields, the framework provides a mechanism to stop listening to a transmission after a specified number of bits have arrived.

Collisions

If a node starts to send a packet while another packet is already being transmitted, this second signal is uncorrelated to the first sender. The framework models this signal as noise and therefore decreases the SNR at the receiver of the first packet. Such events can happen several times during the reception of a data packet, therefore the receiver has to deal with changing SNR throughout the packet receiving process. The final count of bit errors thus results from this sequence of different SNR values and is assembled from the portions of constant SNR. So the bit errors are accumulated for all portions of the packet with constant SNR. The user has to provide the method to calculate the bit error count for a constant portion of SNR, everything else is handled by the framework.

Multiple access

To utilize the RF communication by multiple participants, three multiple access methods are supported. For separate services, it is likely to utilize different frequency bands (e.g., the 2.4 GHz ISM band and the 868 MHz ISM band). Within these bands, a separation using dedicated frequency channels (frequency division multiple access, FDMA) is typically implemented to increase the number of node sending in parallel. The same purpose is served by overlaying the RF signals in time and frequency domain but coding these with different keys (code division multiple access, CDMA). These three multiple access schemes are generalized by the framework. The user provides a function to implement the adjacent channel interference which handles the filter suppression ratio or the coding gain. This is incorporated by the Air to calculate signal and noise power and consequently the SNR. The other two common multiple access formats, space and time division multiple access (SDMA, TDMA) are supported trivially due to the principle of the Air.

3.3.8. Power simulation

A key feature of the framework (regarding PAWiS requirements) is given by the power consumption simulation of tasks. Therefore, a central power meter object logs the power consumption values that are reported by all modules of all nodes. Only tasks that simulate dedicated hardware directly report power consumption. Software tasks report their CPU utilization, and the CPU module calculates its power consumption and reports it on behalf of the requesting task.

Every hardware task that consumes power reports this to the central reporting facility. It can have different electrical behavior, that is, the current I depends in different ways on the supply voltage U. The current can be constant and therefore independent of the supply voltage. A resistive behavior ($I = U/R$) and a combination ($I = I_{\text{const}} + U/R$) can be modeled. Additionally, a user defined, for example, nonlinear characteristic can be implemented and is supported by the framework. The power consumption as well as the electrical behavior can be updated by a task at any point in time.

The reporting of power consumption is accomplished by calls to special methods offered by PAWiS framework classes. The model programmer has to provide the appropriate figures (current, equivalent resistance). These numbers can be determined in several ways. The most accurate numbers result from measurements of real-world devices (e.g., a test chip or prototype PCB) which should be modeled with the PAWiS framework (though this requires the device to be already available). Alternatively the user can obtain the consumption parameters by electrical simulations of the circuitry using, for example, Spice. This is particularly interesting if the model is programmed in parallel to designing the chip of the planned module. For commercially available components (e.g., a microcontroller or RF transceiver) the data sheet provides the appropriate power consumption values.

It is important to mention that modules of a sensor node do not consume constant power throughout their lifetime. On the contrary, the power consumption varies with the operating state. For example, the CPU consumes less power when being in sleep mode (and hence does not execute instructions), the power consumption of the radio differs whether in transmit, in receive, in PLL-locked or in idle mode. The model programmer has to report new power consumption figures every time the state of the module changes.

The framework supports the modeling of a supply hierarchy where the power input of a module (e.g., an ADC), is supplied by the power output of another module (e.g., an LDO). Since a power supply has varying efficiency according to load and input voltage as well as nonzero output resistance, its output voltage and internal consumption are calculated from its output current. This output current is the sum of all supply currents of the supplied modules. Additionally, the framework provides a mechanism to specify different power supply behaviors (particularly the output resistance). This power consumption model results in a simple electrical network.

During the simulation, a task calculates its current, reports this to its power supply and is notified about the actual input voltage by the power supply in return. Most of this is handled automatically by the framework. This mechanism recursively propagates up the supply tree (breadth-first) and is finally reported to the central *power meter* which stores this values to an external data file. In this way for every module and task of every node in the simulated WSN, the power consumption is calculated and logged. With this approach, the power consumption of the whole node is covered and the simulation results plausibly reflect the reality. These results are analyzed and visualized by the data postprocessing tool (see Section 5). So the power simulation values are not actively evaluated during the simulation run but analyzed after the simulation is finished.

3.4. Dynamic behavior

The PAWiS framework supports dynamic behavior (e.g., mobility, environmental dynamics) via an embedded scripting language. Generally, scripting languages are platform independent and need a virtual machine for execution. Although, it degrades execution time compared to compiled languages, it enables code adjustments and algorithm tweaks even at runtime. Scripting languages can be considered as high-level languages as they usually feature dynamic typing, implicit memory management (garbage collection), and often multithreading or support for coroutines intrinsically. An application utilizing an embedded scripting engine needs to provide glue code in order to make internals accessible via scripts. (Glue code is code that does not provide additional functionality but "glues" application specific objects or functions to the scripting engine.) For the PAWiS framework, the scripting language LUA [30] has been chosen due to its simplicity, extensibility, widespread usage, large community, fast execution, and maturity (http://www.lua.org/).

A large portion of the PAWiS framework's basic functionality is glued to the scripting engine. A main part comprises the module's flow control and power consumption functionality which enables the user to provide *functional interfaces* entirely in the scripting language. This is intended to serve as a rapid prototype development scheme without the need to re-compile the entire simulation. Scripts can be hooked to various events fired by the framework, for example, for set up purposes scripts can be hooked to node or network creation events. Usually simple topologies can be setup with the OMNeT++ intrinsic Ned language but more complex topologies can be created utilizing these initialization scripts with less effort. PAWiS scripts can be used to interface with the framework on network-wide and intranode levels during runtime or in the network setup phase.

A real-world scenario for the necessity of introducing dynamic behavior of sensor nodes is shown in Figure 3 where two cars are passing each other. Each car is equipped with a WSN and both networks affect each other when they come close. In order to simulate and observe this simple yet realistic scenario, it is necessary that the two networks can be moved at a certain velocity and in some direction during runtime. This is achieved by grouping the nodes in two distinct networks and then moving these groups via a script in opposite directions. When sensor nodes change their position in space, the PAWiS framework automatically handles the respective change in the network connectivity and the signal strengths without the need of user intervention (i.e., no method needs to be called to update the adjacency matrix). Accordingly, the radio model is rendered dirty and recalculated when the next activity on the air occurs.

Besides mobility, scripting is useful to handle dynamic and reproducible effects that occur within the sensor network's environment [27]. Generally, sensors (even simulated ones) need to monitor a certain phenomenon. Although the PAWiS simulation framework does not provide a detailed model to capture phenomena (e.g., humidity, and luminance), it defines an abstract interface for sensors and the environment. This sensor/phenomenon pair is handled like a typical subscriber/producer pattern. Sensors subscribe to a phenomenon to get it's current value and will be notified of changes instead of frequently polling the phenomenon thus reducing the computational load. The timing behavior of the notification can be set up (e.g., an update frequency or whenever a significant change of a phenomenon's value occurs) for each sensor individually. Sensor properties maintained by the framework are the position of the sensor and its orientation in 3D space. Whenever a sensor reads its associated phenomenon's value, these properties affect the reading as they define the distance and whether the sensor is facing the value's source.

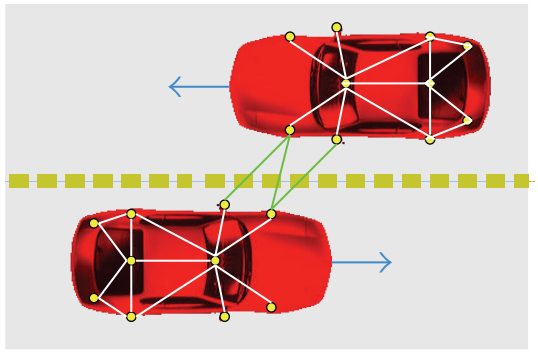


Figure 3: Two cars equipped with wireless sensor networks passing each other.

The semantic of the environmental values has to be introduced by the user of the framework. Though this can be done with C++, it is recommended to use scripts for the environment model as it is more portable and can be exchanged easily for a simulation run. With the help of scripts, even complex environmental scenarios can be easily modeled and simulated. Additionally, the framework supports the usage of prelogged data that can be used as values for phenomena. Currently, effects of interest within the intended field of application for the framework comprise day and night cycles, season changes, weather conditions, and phenomena that support energy scavenging mechanisms.

Utilizing scripting for environment and phenomena as mentioned above introduces reproducible pseudo realistic (opposed to pseudorandom) values to the simulation. While further methods to support more accurate and realistic values exist some of them come up with inherent drawbacks (at least for the targeted field of application). The EmStar [25] framework has the ability to use a hybrid simulation scheme to provide realistic sensor readings. This means that it uses real-sensor readings as input for the simulation. While this brings real-world values into the simulation, it cannot be reproduced over multiple simulation runs. Furthermore, it is not possible to simulate large-time spans, for example, when season-dependent effects need to be considered or the intended network lifetime is up to a year or more.

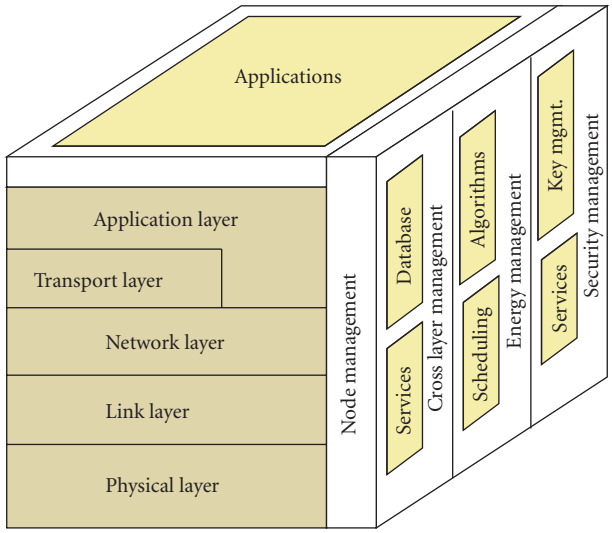


Figure 4: WSN protocol architecture.

4. MODULE LIBRARY

The implementation within a module has to meet the specification of the *functional interfaces* according to the type of the module. This forms another key idea of the modeling process, that is, to provide a library with various distinct implementations for a specific module type. The resulting library can be used to evaluate, refine, and test architectural issues of the user's model. We provide a module library for the proposed protocol architecture (see Section 4.1). These modules can be used in any combination to get an optimized protocol stack for a particular application. Protocol architecture and standardized interfaces (discussed below) make it easy to integrate new modules or interchange different implementations to get optimized results.

4.1. Protocol architecture

The wireless communication characteristics (mobility, rapidly changing link quality, limited resources, and environmental obstructions) and new design paradigms (e.g., wake-up radio) motivate to divert from traditional layered architectures. At the same time "*plug and play*" like features of the layered architecture are important for extensibility and interchangeability.

We define a protocol architecture [31] which provides the benefits of traditional layered architectures [32] and focuses on exploiting synergy across layers (e.g, to extend network life time) [33]. The proposed architecture (see Figure 4) comprises traditional layers (application, transport, network, link, and physical layer) with management planes (cross layer, energy, security, and node). All layers and planes are connected through well-defined interfaces.

The cross-layer management plane (CLAMP) [4] provides a mechanism to exchange cross-layer information but in an optional way that the concept of modularity of layered architectures is still maintained. The CLAMP provides a rich set of parameters available to all the modules of the protocol stack by a publish-notify-update-query mechanism. A discussion of benefits when using cross layer information can be found in [4, 34].

Limited energy sources (e.g., AA battery), insufficient energy from scavenging techniques [35], and the difficulty to replace batteries (cost and geographic reasons) motivated the introduction of an energy management plane (EMP). The EMP can be used to implement algorithms (e.g., [36]) to compute remaining battery capacity or to schedule different events (e.g., updating timers, periodic listening) in order to save energy.

Generally, security is not considered as integrated component of the system architecture (at the start) which affects network security adversely [37]. A security management plane (SMP) is included so that security-related issues (key management algorithms, light-weight encryption and decryption, etc.) can easily be integrated into every component as discussed in [37]. Additionally, network diagnosis and management (NM) used for resetting nodes, remote firmware deployment, address assignment, querying the availability of the node, and so forth, are introduced.

4.2. Interfaces

Standardized interfaces (see Figure 5) between different protocol layers and management planes expedite the process of new module integration and module interchangeability and hence reduces the overall design time. We kept the interfaces small in number and simple to reduce the communication overhead between layers and avoid complexity. Each interface has its own syntax and semantics. The user-defined interfaces can also be used if required. Figure 5 shows the unified view of the protocol layers, management planes, and interfaces among them. The advantage of such interfaces is that it makes the development of protocol layers independent from one another and at the same time provides the functionality to exploit synergy between different layers for cross-layer optimization benefits. Detailed information about the interfaces can be found in [31].

5. DATA POSTPROCESSING

A data postprocessing tool (DPPT) was developed to analyze and visualize the information (power consumption, timing, events, and module interaction) logged during the simulation run. The DPPT (see Figure 6 for a screenshot) along with the PAWiS framework runs on Linux as well as Windows platforms. The DPPT visualizes the simulation results in a hierarchical way by network nodes, modules, submodules, and user-defined categories (e.g., the user can visualize energy consumed by the whole node or by one of its modules like routing or MAC). Each of these elements can be shown or hidden. Elements can be presented in different display colors to be distinguished in the graph. Several navigational helpers (e.g., panning, zooming, scrolling, and snapping) are provided to navigate the simulation data. The DTTP also provides operations on data rows like integration of power consumption, and so forth.

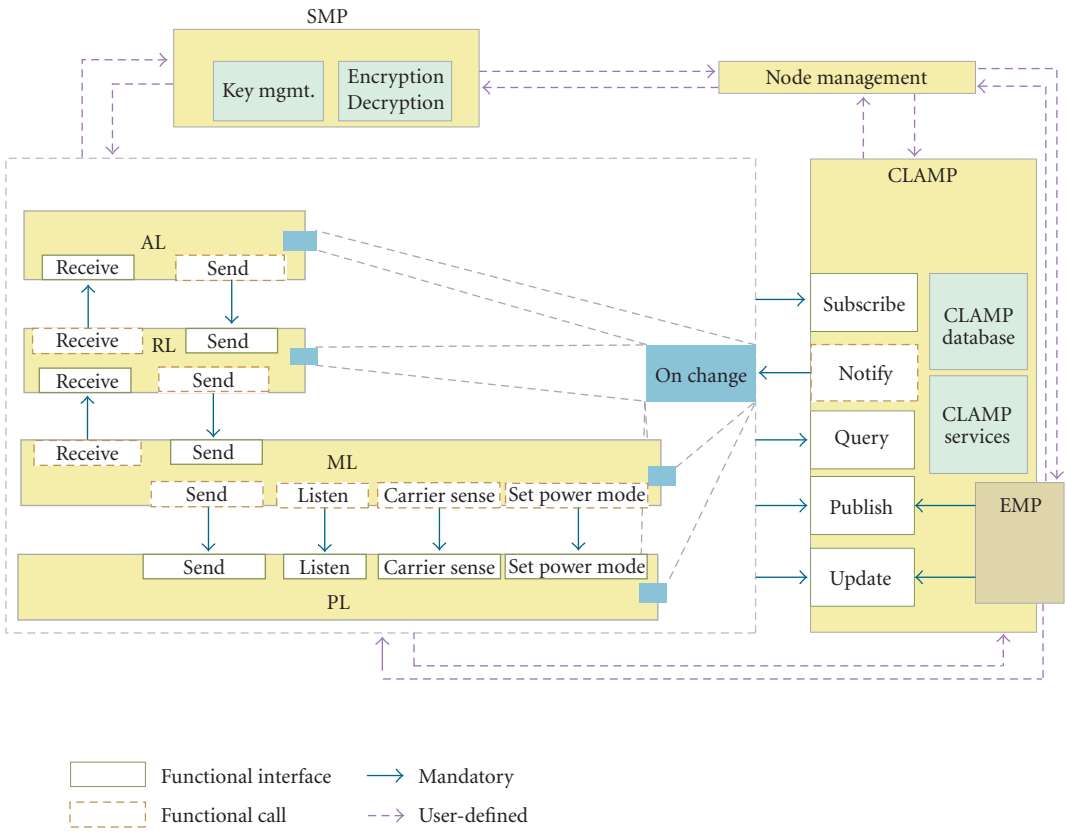


Figure 5: Interfaces between different layers and management planes.

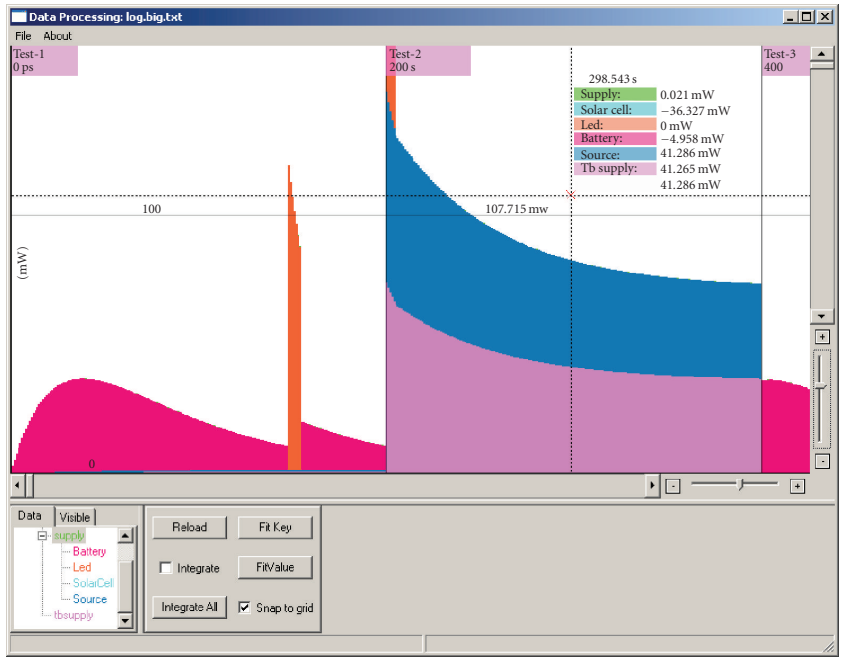


Figure 6: Data postprocessing tool.

The functions of the DPPT are divided into two categories, analysis and visualization of the data. The analysis techniques include the following:

(i) power consumption;

(ii) energy consumption (integral of power consumption over time);

(iii) reveal network topology from routing tables;

(iv) calculate remaining battery capacity from power consumption;

(v) sum of power consumption of a certain module type of all nodes;

(vi) statistical analysis of end-to-end packet delay, number of retransmissions, unnecessary CPU wakeups, and more;

(vii) power and/or energy consumption distribution across several nodes.

The analysis results are visualized by the following:

(i) chart (pie, bar, stack, etc.);

(ii) tables (showing numbers);

(iii) 3D maps (power and/or energy consumption distribution);

(iv) vectors (network topology, routing decisions).

The main display as shown in Figure 6 is the power consumption of the modules of a node plotted over time. The individual power values are stacked and colored according to the module. As the user moves the mouse, an overlay shows the power values at the cursor.

One major difficulty for displaying the data is the high dynamic range of values and times. For WSN nodes, it is typical that within a very short time interval many events (and therefore changes in power consumption) occur followed by a long period of no actions. The user would have to constantly zoom in and out to inspect the points of interest. The same problem applies to power consumption values, for example, the power consumption of the RF transceiver is higher than the consumption of the AD converter by a factor of several hundreds. We tackle this visualization problem by applying a non-linear monotonic concave function (e.g., the logarithm) to the time and/or value *differences*. (We do not transform the absolute time and sum of power values (as for logarithmic plots) but the differences from one discrete event to the next.) This reduces the dynamic range of the differences while equal time intervals and power values are displayed equally at every absolute point.

6. RESULTS

In this section, we present a performance evaluation of the PAWiS framework as well as a case study to show its benefit in designing, simulating, and optimizing real-world scenarios.

Table 1: Performance results for different network sizes.

Nodes	Sim s/s	Event/s
20	207.6790	32,808
50	52.8075	23,975
100	7.9105	16,666
200	2.6811	9,756
500	0.2249	4,025
1,000	0.1278	2,298

6.1. Performance evaluation

To get performance figures, we abstracted an application layer by probabilistic sampling based on uniformly distributed intervals (a node waits between 20 to 70 seconds and then sends data with a probability of 20%). A position-based routing scheme (a scheme which forwards data based on location of the immediate source, the node itself, and that of the destination node) which maximizes progress (minimizes distance) towards the destination node was implemented on the routing layer. At the MAC layer, a simple CSMA/CD (carrier sense multiple access with collision detection) scheme was implemented. A linear battery discharge model and a first-order radio model was used. Many to one communication was considered where all the nodes send data to the sink node located at the center of the network area. Multiple simulation runs with various network sizes and approximately equal node density were made for the performance evaluation. The network nodes were distributed on a grid with a jitter around the grid positions. Scripting was only used to set up the network and simulation environment, but no dynamic effects during simulation runtime have been implemented. The used simulation model put more emphasis on higher-level effects (on protocol level) than on intranode communication or instruction set simulation. For the execution of each simulation run, a common desktop PC equipped with an AMD Athlon 64 3000 CPU (2.0 GHz) and 1.5 GB of RAM has been used.

The simulation framework generates messages for a lot of events that occur during simulation. These messages are dispatched by the OMNeT++ simulation kernel. Therefore, the number of messages (events) generated during a simulation run is a key figure of the performance evaluation. An additional important figure constitutes the ratio between simulated seconds and wall-clock seconds for each run. These figures were acquired by simulating one million events for different network sizes.

Table 1 shows the simulation results for 20, 50, 100, 200, 500, and 1000 nodes. It should be noted that for this particular simulation configuration, the ratio between simulation time and wall-clock time is just an example and should not be seen as a general figure for any simulation. More important are the figures regarding the processed events per second. As the framework handles many aspects of wireless communication, the duration for processing events with lots of nodes increases significantly. This is

related to the fact that the more nodes are simulated, the more signal-dependent calculations have to be computed at each sending operation thus resulting in less processed events per second. Evidently, the node's granularity (in terms of hardware and software) additionally affects the overall simulation performance. The finer grained a node is, the more events are needed to simulate one node which further drops the ratio between simulation time and wall-clock time.

In principle, a finer model granularity results in an increased number of OMNeT++ modules and submodules. OMNeT++ supports parallel discrete event simulation (PDES) strategies to enhance overall simulation performance but special measures need to be taken to ensure the synchronization of modules on distinct processors thus reducing the local simulation throughput. However to benefit from PDES, the model needs to consider some message exchange constraints to ensure the causality of events on distinct processors. Currently, the PAWiS framework does not comply with all of these constraints. Therefore, it is not yet possible to utilize PDES in order to speed up the simulation (though it is planned to make future versions of the framework PDES compliant).

The performance figures for this particular simulation show that with a network size of 500 nodes the simulation time progresses slower than wall clock time which makes it rather impractical to simulate larger networks over long-time periods (e.g., simulating 30 days for 1000 nodes would take 235 days!). As expected, these figures suggest that the framework needs to process more events per second and node with increasing network sizes. Though these figures seem to be not very promising, it should be noted that simulating this large networks at fine granularity is of little practical use. If one wants to simulate at fine granularity within one node, smaller network sizes also help identifying functionality issues and inefficiencies. In a second step larger networks may be simulated by omitting implementation details which helps to reduce the number of events. Moreover, simulating networks with sizes up to 30 nodes does better reflect currently commercially developed sensor network applications.

A scalability evaluation in [13] targeting various aspects of WSN simulation (some of them comparable to the PAWiS framework) with OMNeT++ confirms the above-mentioned figures and conclusions. The authors showed performance figures for WSN models with different complexities and sizes (a full model contains radio, environment, and battery) and finally conclude that simulating larger networks with current approaches is not practical.

6.2. Case study

A case study of a WSN for a tire pressure monitoring system (TPMS) in automobiles (as depicted in Section 3.4 and Figure 3) has been modeled and simulated with the PAWiS framework. The TPMS model consists of four wireless sensor nodes and a sink node. Each sensor node is attached to a different wheel while the sink node is placed in the central console. All sensor nodes are battery powered whereas the sink is connected to the car-power supply. Each sensor periodically (every 20 seconds) reads the sensor values, and in case of a significant change from previous readings (this is modeled with a sending probability of 10%) it sends a packet to the sink node. As the focus of this example is not power awareness, the CPU never enters sleep mode but permanently executes code from the application. If the sink node receives the packet without bit errors, it sends an acknowledgment back to the originator node. At most, three retransmissions are attempted in case of acknowledgment timeouts by the sender. A sensor node in the model consists of modules for application, MAC and routing, timer, and radio. The application module is further divided into a sensor and an ADC for acquiring the tire pressure. In this model, MAC (utilizing CSMA strategy) and routing are combined as there is no explicit routing needed. The simulation of this TPMS model shows the power consumption of each of these modules. Resulting data can be further analyzed to figure out major power consuming modules and optimize them in order to increase the network life time. Two application scenarios have been simulated: the first scenario with a single car (consisting of the four sensor nodes and one sink node) and a second scenario including two cars within each other's proximity to observe the differences in power profiles of different modules. The two-car scenario should show the impact regarding power consumption and packet delivery ratio of potential interferers (packets from the opposing car are treated as noise) on the local sensor network in each car. After simulating both scenarios, the aforementioned DPPT (see Section 5) was used to integrate the power consumption figures of all the modules. The results are shown in Figure 7 where the power consumption of a specific sensor node from both scenarios is presented. As expected, the major part of power is consumed by the application module (indirectly as the CPU is actually consuming the power for software modules) with almost the same value for both scenarios. The sensor/ADC and timer modules contribute only with very small amounts to the overall power consumption since they are inactive for a significant portion of time. In the single-car scenario, the radio and MAC/routing consumed significantly less power than in the two-car scenario. This happens due to the fact that an increased number of sensor nodes within the vicinity of each other results in more network collisions which raise the bit error probability. Furthermore, bit errors result in packet retransmissions which additionally keep the radio in listening state for a longer time to wait for acknowledgments. The same applies for the MAC as more occurrences of timeouts and retransmits have to be processed.

We simulated this simple and straightforward case study of a TPMS on purpose. It took only about 20 man-hours to model and simulate the real world scenario from scratch. Different scenarios were simulated (e.g., without acknowledgments, moving cars, static single car, etc.) with minor adjustments in the configuration file in less time. With the framework, module inefficiencies can easily be identified (the absolute figures may not be entirely accurate, although relative information may provide a deep insight) and can also be corrected in very short development cycles.

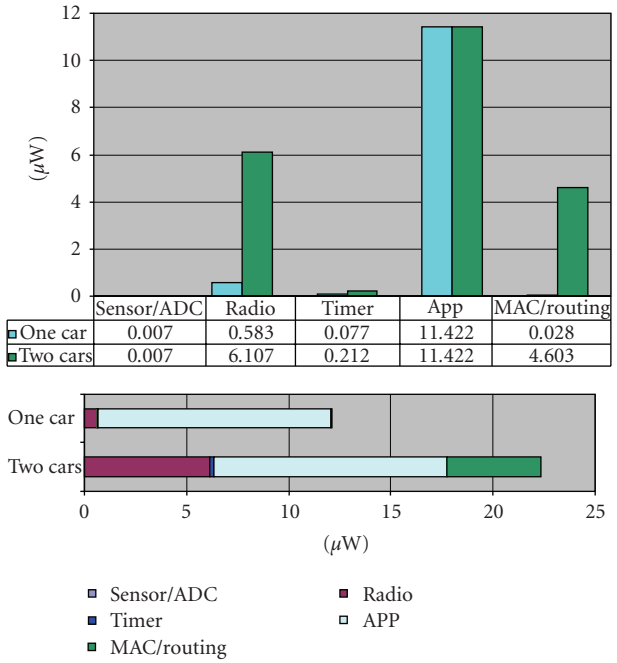


Figure 7: Power consumption results of the TPMS simulation for one node.

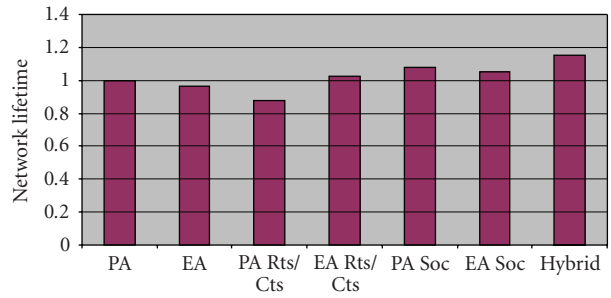


Figure 8: Network life time for various position-based routing implementations.

6.3. Optimization study

In this case study, we discuss the optimization methodology for the development of a routing protocol with the PAWiS framework. This study was aimed to develop and optimize a table-less position-based routing protocol. Initially, we created a skeleton model (the same as used for the study in Section 6.1) for a simple node consisting of application layer (probabilistic sampling), MAC layer (CSMA), and physical layer (simple radio) as well as an implementation of the initial version of a table-less routing scheme which we call progress aware (PA). As the composition of the protocol stack is easily managed with the configuration file without the need to recompile the particular simulation, we executed many trials with numerous node compositions. After analyzing the simulation results, we identified several enhancements and alternatives for various aspects of the routing protocol. Consequently, multiple refinement cycles with distinct implementations of the position based routing strategy were applied. These strategies include PA, energy aware (EA), PA and EA with Rts/Cts packet exchanges, PA and EA with state of charge (SOC) packet exchanges, and even hybrid approaches (to dynamically change between progress aware and energy aware routing) to route packets. First, the performance figures acquired in each refinement iteration were analyzed to identify the main contributors to power consumption in the routing layers. In the next step, the gained insights were reapplied to the routing implementations resulting in additional refinement/analysis cycles until the results met the targeted power consumption constraints. The chart in Figure 8 depicts the network lifetime for various implementations of position-based routing compared to our initial version of a protocol that utilized a PA scheme (with network lifetime scaled to 1 in the chart). The chart shows that the EA scheme exhibits lower lifetime compared to the PA scheme which can be perceived as a contradiction. The reason for this is that in this scheme packet forwarding is based on timers. For example, assume a four-node network comprising a source node S, a destination node D and two potential forwarders A and B. If S has some data to transmit, it sends them blindly. A and B being within transmission range of S receive them and start their respective timers. The timer of the node which provides maximum progress expires first (i.e., the timer of the node with more remaining energy), and further forwards the data while the other node upon listening to that data kills its timer and drops that data. EA scheme always try to balance energy consumption across nodes by routing through nodes with more remaining energy, therefore, once energy balance (all nodes having almost equal remaining energy) across the network is achieved, duplicate packets occur as timers of potential forwarders expire at the same time. These duplicate packets cause the routing layer to utilize the radio more often and hence drain the battery quicker which results in reduced network lifetimes.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a power aware discrete event simulation framework for wireless sensor networks and nodes. Based on OMNeT++, it provides additional features to capture energy consumption (at any desired level), introducing a model for the RF communication (enabling complex topologies) and environmental effects with scripting capabilities. It provides a visualization tool to analyze and visualize energy usage. The framework focuses on extensibility and reusability by outlining a protocol architecture and provides module library. The results show that the performance (execution time) of the PAWiS simulation framework is comparable with other frameworks and appropriate for the targeted field of application. The case studies showed that the modularization of OMNeT++ models combined with the abstract component concept of the PAWiS framework generally results in a reduced design-debug cycle.

The framework in its current version handles already much of the functionality and effects that are important for

simulating wireless sensor networks. However, there are still some extensions and features that need to be enhanced or included. Performance of the simulation framework needs to be further enhanced, as the current results show scalability issues for larger networks (see Section 6.1).

Furthermore, an important aspect of the simulation framework is the postprocessing tool chain. The visualization tool will be extended by additional functions for analyzing the output of the simulation application. Currently, the output comprises power consumption, fired events, and module occupation. Additionally, analysis tools will be included to allow the comparison of nodes regarding various properties gained from the simulation and features like significant power peak detection, min-max over sliding window, integration features.

Additional and updated information regarding the PAWiS project and the simulation framework is available at http://www.ict.tuwien.ac.at/pawis/.

REFERENCES

[1] D. Weber, J. Glaser, and S. Mahlknecht, "Discrete event simulation framework for power aware wireless sensor networks," in *Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN '07)*, vol. 1, pp. 335–340, Vienna, Austria, June 2007.

[2] J. Glaser and D. Weber, "Simulation framework for power aware wireless sensors," in *Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN '07)*, K. Langendoen and T. Voigt, Eds., pp. 19–20, Delft, The Netherlands, January 2007.

[3] S. Mahlknecht, J. Glaser, and T. Herndl, "PAWiS: towards a power aware system architecture for a SoC/SiP wireless sensor and actor node implementation," in *Proceedings of 6th IFAC International Conference on Fieldbus Systems and Their Applications*, pp. 129–134, Puebla, Mexico, November 2005.

[4] S. A. Madani, S. Mahlknecht, and J. Glaser, "CLAMP: cross layer management plane for low power wireless sensor networks," in *Proceedings of the 5th International Workshop on Frontiers of Information Technology*, Islamabad, Pakistan, December 2007.

[5] E. Egea-López, J. Vales-Alonso, A. Martínez-Sala, P. Pavón-Marñio, and J. García-Haro, "Simulation tools for wireless sensor networks," in *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '05)*, Philadelphia, Pa, USA, July 2005.

[6] S. Kurkowski, T. Camp, and M. Colagrosso, "MANET simulation studies: the incredibles," *Mobile Computing and Communications Review*, vol. 9, no. 4, pp. 50–61, 2005.

[7] D. Wetherall and C. J. Lindblad, "Extending Tcl for dynamic object-oriented programming," in *Proceedings of the 3rd USENIX Annual Tcl/Tk Workshop*, p. 19, Toronto, Canada, July 1995.

[8] V. Naoumov and T. Gross, "Simulation of large ad hoc networks," in *Proceedings of the 6th ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems (MSWiM '03)*, pp. 50–57, San Diego, Calif, USA, September 2003.

[9] G. Chen, J. Branch, M. Pflug, L. Zhu, and B. K. Szymanski, "Sense: a wireless sensor network simulator," in *Advances in Pervasive Computing and Networking*, chapter 1, pp. 249–267, Springer, New York, NY, USA, 2006.

[10] S. Park, A. Savvides, and M. B. Srivastava, "SensorSim: a simulation framework for sensor networks," in *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '00)*, pp. 104–111, Boston, Mass, USA, August 2000.

[11] A. Varga, "The OMNeT++ discrete event simulation system," in *Proceedings of the 15th European Simulation Multiconference (ESM '01)*, pp. 319–324, Prague, Czech Republic, June 2001.

[12] W. Drytkiewicz, S. Sroka, V. Handziski, A. Koepke, and H. Karl, "A mobility framework for OMNeT++," in *Proceedings of the 3rd International OMNeT++ Workshop*, Budapest, Hungary, January 2003.

[13] E. Egea-López, J. Vales-Alonso, A. Martínez-Sala, P. Pavón-Marñio, and J. García-Haro, "Simulation scalability issues in wireless sensor networks," *IEEE Communications Magazine*, vol. 44, no. 7, pp. 64–73, 2006.

[14] C. Mallanda, A. Suri, V. Kunchakarra, S. S. Iyengar, R. Kannan, and A. Durresi, "Simulating wireless sensor networks with OMNeT++," submitted to *IEEE Computers*.

[15] X. Zeng, R. Bagrodia, and M. Gerla, "GloMoSim: a library for parallel simulation of large-scale wireless networks," in *Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS '98)*, pp. 154–161, Banff, Canada, May 1998.

[16] M. Varshney, D. Xu, M. B. Srivastava, and R. L. Bagrodia, "SenQ: a scalable simulation and emulation environment for sensor networks," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*, pp. 196–205, Cambridge, Mass, USA, April 2007.

[17] C. Perkins and E. Royer, "Ad hoc on-demand distance vector routing," in *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, pp. 90–100, New Orleans, La, USA, February 1999.

[18] D. B. Johnson and D. A. Maltz, "Dynamic source routing in ad hoc wireless networks," in *Mobile Computing*, Kluwer Academic Publishers, Norwood, Mass, USA, 1996.

[19] D. Cavin, Y. Sasson, and A. Schiper, "On the accuracy of MANET simulators," in *Proceedings of the 2nd ACM International Workshop on Principles of Mobile Computing (POMC '02)*, pp. 38–43, Toulouse, France, October 2002.

[20] G. Chen and B. K. Szymanski, "COST: a component-oriented discrete event simulator," in *Proceedings of the 34th Winter Simulation Conference (WSC '02)*, vol. 1, pp. 776–782, San Diego, Calif, USA, December 2002.

[21] J. Liu, X. Liu, and E. A. Lee, "Modeling distributed hybrid systems in Ptolemy II," in *Proceedings of the American Control Conference*, vol. 6, pp. 4984–4985, Arlington, Va, USA, June 2001.

[22] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao, "Modeling of sensor nets in Ptolemy II," in *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks (IPSN '04)*, pp. 359–368, Berkeley, Calif, USA, April 2004.

[23] J. A. Miller, R. S. Nair, Z. Zhang, and H. Zhao, "JSIM: a Java-based simulation and animation environment," in *Proceedings of the 30th Annual Simulation Symposium*, pp. 31–42, Atlanta, Ga, USA, April 1997.

[24] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire TinyOS applications," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*, pp. 126–137, Los Angeles, Calif, USA, November 2003.

[25] L. Girod, T. Stathopoulos, N. Ramanathan, et al., "A system for simulation, emulation, and deployment of heterogeneous sensor networks," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pp. 201–213, Baltimore, Md, USA, November 2004.

[26] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, and M. Karir, "ATEMU: a fine-grained sensor network simulator," in *Proceedings of the 1st Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON '04)*, pp. 145–152, Santa Clara, Calif, USA, October 2004.

[27] M. Demmer, P. Levi, A. Joki, E. Brewer, and D. Culler, "Tython: a dynamic simulation environment for sensor networks," Tech. Rep. UCB/CSD-05-1372, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Calif, USA, 2005.

[28] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pp. 188–200, Baltimore, Md, USA, November 2004.

[29] J. Heidemann, N. Bulusu, J. Elson, et al., "Effects of detail in wireless network simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 3–11, USC/Information Sciences Institute, Society for Computer Simulation, Phoenix, Ariz, USA, January 2001.

[30] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua—an extensible extension language," *Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[31] S. A. Madani, S. Mahlknecht, and J. Glaser, "A step towards standardization of wireless sensor networks: a layered protocol architecture perspective," in *Proceedings of the International Conference on Sensor Technologies and Applications*, pp. 82–87, Valencia, Spain, October 2007.

[32] V. Kawadia and P. R. Kumar, "A cautionary perspective on cross-layer design," *IEEE Wireless Communications*, vol. 12, no. 1, pp. 3–11, 2005.

[33] W. Su and T. L. Lim, "Cross-layer design and optimization for wireless sensor networks," in *Proceedings of the 7th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD '06)*, pp. 278–284, Las Vegas, Nev, USA, June 2006.

[34] V. T. Raisinghani and S. Iyer, "Cross-layer design optimizations in wireless protocol stacks," *Computer Communications*, vol. 27, no. 8, pp. 720–724, 2004.

[35] S. Roundy, D. Steingart, L. Frechette, P. Wright, and J. Rabaey, "Power sources for wireless sensor networks," in *Proceedings of the 1st European Workshop on Wireless Sensor Networks (EWSN '04)*, vol. 2920, pp. 1–17, Springer, Berlin, Germany, January 2004.

[36] D. Rakhmatov, S. Vrudhula, and D. A. Wallach, "A model for battery lifetime analysis for organizing applications on a pocket computer," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 1019–1030, 2003.

[37] A. Perrig, J. Stankovic, and D. Wagner, "Security in wireless sensor networks," *Communications of the ACM*, vol. 47, no. 6, pp. 53–57, 2004.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 647953, 13 pages
doi:10.1155/2008/647953

Research Article

System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design

Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D. Gajski

Center for Embedded Computer Systems, University of California, Irvine, CA 92697-2625, USA

Correspondence should be addressed to Rainer Dömer, doemer@uci.edu

Received 1 October 2007; Revised 4 March 2008; Accepted 10 June 2008

Recommended by Christoph Grimm

The constantly growing complexity of embedded systems is a challenge that drives the development of novel design automation techniques. C-based system-level design addresses the complexity challenge by raising the level of abstraction and integrating the design processes for the heterogeneous system components. In this article, we present a comprehensive design framework, the system-on-chip environment (SCE) which is based on the influential SpecC language and methodology. SCE implements a top-down system design flow based on a specify-explore-refine paradigm with support for heterogeneous target platforms consisting of custom hardware components, embedded software processors, dedicated IP blocks, and complex communication bus architectures. Starting from an abstract specification of the desired system, models at various levels of abstraction are automatically generated through successive step-wise refinement, resulting in a pin-and cycle-accurate system implementation. The seamless integration of automatic model generation, estimation, and verification tools enables rapid design space exploration and efficient MPSoC implementation. Using a large set of industrial-strength examples with a wide range of target architectures, our experimental results demonstrate the effectiveness of our framework and show significant productivity gains in design time.

1. INTRODUCTION

The rising complexity of embedded systems challenges the established design techniques and processes. Novel, nontraditional design approaches become necessary in order to keep up with the increasing demands of higher productivity.

A well-known technique to address the system design challenge is system-level design which raises the level of abstraction, exploits the reuse of intellectual property (IP), and integrates the traditionally separate design processes of the heterogeneous system components. By combining the design flows of hardware units, software processors, third-party IPs, and the interconnecting bus architectures, system-level design emphasizes the system perspective of the overall design task and enables design space exploration across domains. However, successful system design depends on efficient design automation techniques and, in particular, effective tool support.

In this article, we describe the system-on-chip environment (SCE), a system-level design framework based on the SpecC language and methodology [1]. SCE realizes a top-down refinement-based system design flow with support of heterogeneous target platforms consisting of custom hardware components, embedded software processors, dedicated IP blocks, and complex communication bus architectures.

1.1. SCE methodology

Figure 1 shows the design flow with SCE in an overview. Starting with an abstract specification model in the system design phase, the designer automatically generates transaction level models (TLM) of the design, successively at lower levels of abstraction. Based on component models from the system database and design decisions made by the user, the generated models carry an increasing amount of implementation details.

SCE follows a *specify-explore-refine* methodology [2]. The design process starts from a model specifying the design functionality (*specify*). At each following step, the designer first explores the design space (*explore*) and makes the

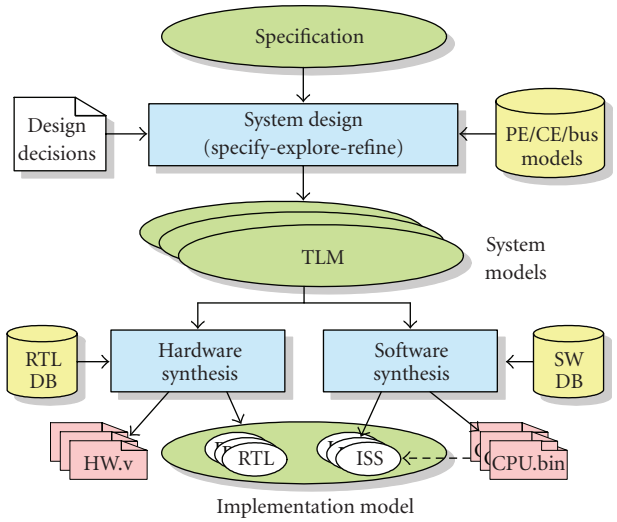
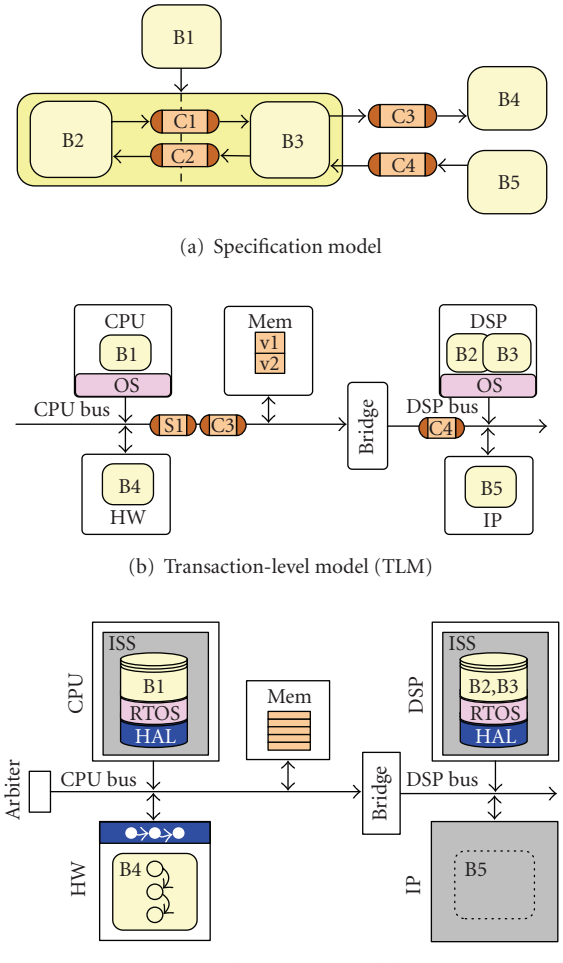


Figure 1: System-on-chip environment (SCE) design flow.



(a) Specification model

(b) Transaction-level model (TLM)

(c) Implementation model

Figure 2: Generic SCE design models.

necessary design decisions. SCE then automatically generates a new model by integrating the decisions into the previous model (*refine*).

After the system design phase is complete, the hardware and software components in the system model are implemented by the hardware and software synthesis phases, respectively. As a combined result, a pin- and cycle-accurate implementation model is generated. Also, binary images for the software processors, as well as register-transfer level (RTL) descriptions in Verilog for the hardware blocks, are created for further synthesis and manufacturing of the intended Multiprocessor system-on-chip (MPSoC).

Three design models used in the SCE design flow are shown in more detail in Figure 2. In these and later figures that describe design models, we use the graphical notation introduced with the SpecC language [1]. In general, rectangular boxes represent components, and interconnections are indicated by lines (wires) and arrows (busses). Encapsulated computational blocks, called behaviors, are shown as rectangular boxes with round corners, whereas high-level communication is indicated by channels (ellipses) and interfaces (half circles).

Figure 2(a) depicts a simple generic specification model. The model consists of a hierarchy of five behaviors and four communication channels. Except for the system functionality, this model is free of any implementation details. During the system design phase, it will be mapped to a platform architecture (see Section 3.1) and single-threaded processing elements (PEs) will be scheduled (Section 3.2). Communication elements (CEs), such as bus bridges and transducers, and system busses will be added to the model as well (Sections 3.3 and 3.4).

As a result of each of these model refinement steps, a TLM is generated, as shown in Figure 2(b). Depending on the number of implementation decisions taken, the TLM accurately reflects the number and type of PEs in the architecture, the mapping of behaviors to the PEs, and the mapping of channels to the system busses and CEs. Note that the communication in this model is still at the abstract transaction level.

After hardware and software synthesis (see Sections 3.5 and 3.6, resp.), a cycle-accurate implementation model is generated, as illustrated in Figure 2(c). In this model, embedded software is represented in detailed layers, including the real-time operating system (RTOS) and the hardware abstraction layer (HAL). Custom hardware blocks, on the other hand, are represented accurately by RTL finite state machine (FSM) models. Finally, system communication is also refined down to a pin- and cycle-accurate level.

1.2. Related work

Traditionally, system design is dominated by simulation-centric approaches with horizontal integration of models at specific levels of abstraction. Approaches range from the cosimulation of different low-level languages [3–5] to the combination of heterogeneous models of computation in a common simulation environment [6]. In between, C-based system-level design languages (SLDLs), such as SystemC [7]

and Handel-C [8], emerged as vehicles for transaction-level modeling (TLM) [9]. Most cases, however, are limited to simulation only and lack vertical integration with synthesis flows that provide a path to implementation.

The first attempts at providing system design environments were approaches for hardware/software codesign. Examples of such environments include COSYMA [10], COSMOS [11], and POLIS [12]. These approaches, however, are based on architecture templates consisting of a single microcontroller assisted by a custom hardware coprocessor, and are thus limited to narrow target architectures.

More recently, design environments emerged that provide support for more complex multiprocessor systems. The OCAPI system [13, 14] is based on an object-oriented modeling of designs using a C++ class library and focuses on reconfigurable hardware devices (FPGAs). The OSSS methodology [15] defines an automated system design flow from a cycle-accurate specification written in an object-oriented variant of SystemC. Supporting architecture exploration and automated refinement via intermediate design models, OSSS feeds into the FOSSY synthesis tool for implementation in hardware and software.

Around the TLM concept, several SystemC-based approaches exist that deal with assembly, validation and to some extent automatic generation of communication [16–20]. Metropolis [21, 22] is a modeling and simulation environment based on the platform-based design paradigm. The key idea is to separate function, architecture, and model of computation into separate models. Although Metropolis allows cosimulation of heterogenous PEs as well as different models of computation, a refinement or verification flow between different abstraction levels has not emerged. None of the above frameworks provides a comprehensive, automated approach for the design of complete MPSoCs from abstract specification down to final implementation.

SCE was built on experiences obtained from its predecessor, SpecSyn [2]. While SpecSyn was based on the SpecCharts language, an extension of VHDL, SCE is based on SpecC, which extends ANSI-C for hardware and system modeling.

With respect to our previous publications (previous publications focus on point-tools within the SCE environment and are referenced where applicable), this article is the first comprehensive, cohesive, and complete description of the SCE framework. In other words, for the first time we describe the entire SpecC methodology as implemented by a real working environment. As such, this article focuses on the integration of the tools (including scripting facilities, file formats, and annotations) that realize an efficient top-down system design flow, all the way from an abstract system specification down to a pin- and cycle-accurate implementation. We also list the design decisions taken at each step and thus provide a complete picture of the input the system designer needs to provide based on his application knowledge and design experience. We also demonstrate the effectiveness of the SCE framework and the complete design flow using the combined results of six design experiments using real-world examples. Furthermore, this article describes for the first time the integration of verification tools into the design flow and framework.

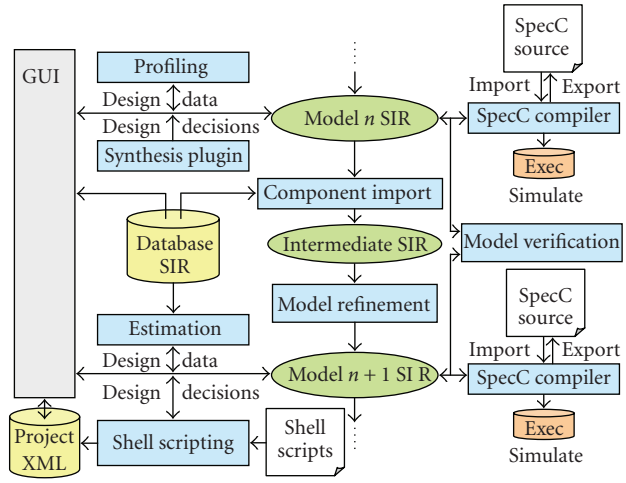


Figure 3: SCE software architecture.

2. SCE ARCHITECTURE

SCE is based on the separation of design tasks into two distinct steps: decision making and model refinement. Model refinement takes design decisions and generates a new model of the design reflecting and implementing the decisions.

In SCE, model refinement is automated. Decisions, on the other hand, can be entered manually or through a tool box of automated synthesis algorithms. Together, SCE supports an interactive and automated system design process. Automatic model generation removes the need for error-prone and tedious model rewriting. Instead, designers can focus on design exploration and decision making.

Figure 3 shows the generic software architecture for each task in the SCE design and refinement flow. In each step, design decisions are entered by the user through a graphical user interface (GUI), via a command-line scripting and shell interface, or with the help of automated synthesis plugins implementing optimizing algorithms. Based on the design decisions, a refinement process generates a new design model from the input model automatically.

Overall, the SCE framework is formed by the combination of point tools. These tools exchange information through command line interfaces and design models. In general, all tools operate on a given design model. Design decisions, profiling data, and metainformation about the design are stored as annotations attached to the corresponding objects in the design and database models. All models and databases in SCE are described and captured in the form of SpecC internal representation (SIR) files. Using the SpecC compiler (`scc`), SCE models and databases can be imported from and exported into source files in standard SpecC language format at any time.

2.1. Graphical user interface

The main interface between the designer and the tools is the `sce` GUI [23] which provides various displays and dialogs for browsing of design models and databases, interactive

decision entry, and graphical analysis of profiling and estimation results. Furthermore, it includes menus and tool bars to trigger simulation, profiling, refinement, synthesis, and verification actions. For each action, specific command-line tools are called and executed as needed where the GUI supplies the necessary parameters, captures the output and handles (normal or abnormal) results.

In each session, multiple candidate designs and models can be explored and generated. Information about design models and their relationships, including project-specific compiler and simulator parameters, are tracked by the GUI and can be stored in project files in a custom XML format, allowing for persistent storage, documentation, and exchange of metainformation about the exploration process.

2.2. Simulation and profiling

All design models in the SCE flow are executable for validation through simulation. Using the SpecC compiler and simulator, models can be compiled and executed at any time. SCE also includes profiling tools to obtain feedback about design quality metrics. Based on a combination of static and dynamic analysis, a retargetable profiler (`scprof`) provides a variety of metrics across various levels of abstraction [24]. Initial dynamic profiling derives design characteristics through simulation of the input model. The system designer chooses a set of target PEs, CEs, and busses from the database, and the tool then combines the obtained profiles with the characteristics of the selected components. Thus, SCE profiling is retargetable for static estimation of complete system designs in linear time without the need for time consuming resimulation or reprofiling.

The profiling results can also be back-annotated into the output model through refinement. By simulating the refined model, accurate feedback about implementation effects can then be obtained before entering the next design stage.

Since the system is only simulated once during the exploration process, the approach is fast yet accurate enough to make high-level decisions, since both static and dynamic effects are captured. Furthermore, the profiler supports multilevel, multimetric estimation by providing relevant design quality metrics for each stage of the design process. Therefore, profiling guides the user in the design process and enables rapid and early design space exploration.

2.3. Verification

SCE also integrates a formal verification tool `scver`. Our equivalence verification technology is based on *model algebra* [25], which is a formalism for symbolic representation and transformation of system level models. The formalism itself consists of a set of objects and composition rules. The objects are behaviors, synchronization channels, variables, and ports. The composition rules for control flow, blocking, and nonblocking communication, and hierarchy allow creation of formal models. Functionality preserving transformation rules are also defined on model algebraic expressions. Each of these transformation rules are proven sound with respect to a trace-based notion of functional equivalence.

The incorporation of model algebra-based verification in SCE follows the refinement flow. Well-formed models in SpecC can easily be translated to respective model algebraic expressions. The system designer simply selects an original and a refined model and invokes the verification tool. `scver` then converts the models and applies the transformation rules to derive the refined model from the original model. The two models are equivalent by virtue of the soundness of the transformation rules. The original model is then checked for isomorphism against the derived model and the differences, if any, are reported. It must be noted that the number and order of transformation rules used for the model derivation step depend on the type of refinement. Since the key concept in SCE is the well-defined semantics of models at different abstraction levels, the order of transformation rules can be easily established. Therefore, equivalence verification becomes not only tractable, but straightforward.

2.4. Databases

In the SCE design flow, the system is gradually refined using system components from a set of databases [26]. Specifically, SCE includes databases for processing elements (PEs), communication elements (CEs), operating system models, bus or other communication protocols, RTL units and software components. The database components are described as SpecC objects (behaviors or channels). The SpecC hierarchy for a component object in the database defines its structure and functionality for simulation and synthesis. In addition, metadata, such as attributes, parameters, and general information, is stored in the form of annotations attached to the components.

2.5. Scripting interface

SCE supports scripting of the complete environment from the command line without the need for the GUI. For scripting purposes, a GUI-less command shell, `scsh`, of SCE is available. The SCE shell is based on the same libraries as the SCE GUI (not including the GUI layer itself) and offers interactive command-prompt based- or automatic script-based execution.

The SCE shell is based on an embedded Python interpreter that is extended with an API for low-level access to SCE core functionality and internals. For user-level scripting, a complete set of high-level tools on top of the SCE shell are available. Provided scripts include command-line utilities for component allocation (`sce_allocate`), mapping/partitioning (`sce_map`), scheduling (`sce_schedule`), connectivity definition (`sce_connect`), component import (`sce_import`), and project handling (`sce_project`). These scripts provide a convenient command-line interface for all SCE high-level functionality and decision entry. Together with command-line interfaces to refinement tools and the compiler, a complete scripting of the SCE design flow, through shell scripts or via Makefiles, is available.

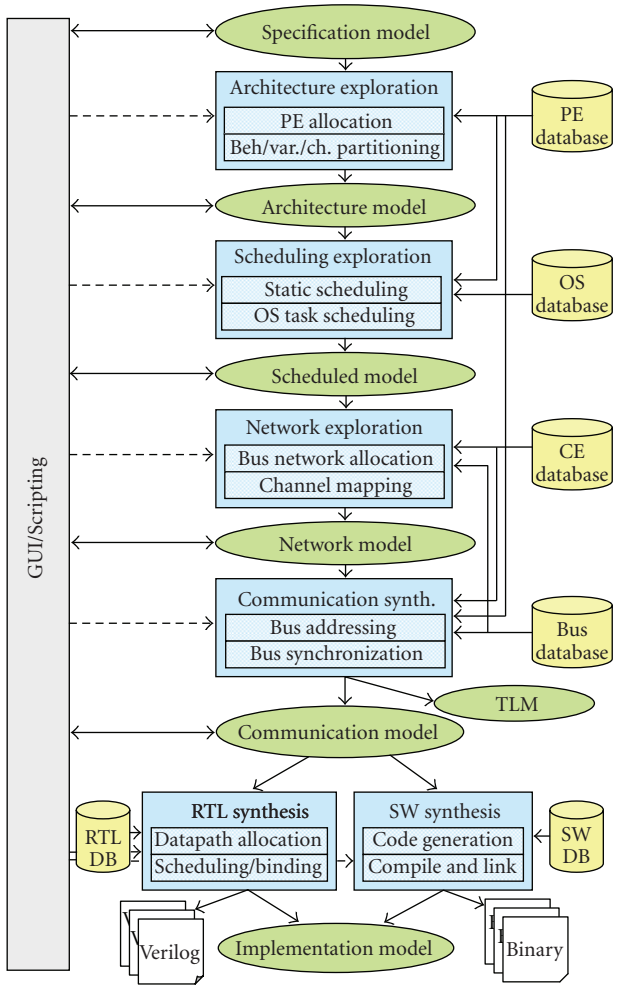


Figure 4: Refinement-based tool flow in SCE.

3. SCE DESIGN FLOW

Figure 4 shows the refinement-based tool flow in SCE from the initial abstract specification down to the final implementation model. In particular, the SCE flow consists of six specific tools which we will describe in the following sections.

3.1. Architecture exploration

The first step in the SCE design flow, architecture exploration, defines the target platform and, under a set of design constraints, maps the computational parts of the specification model onto that platform. The target architecture consists of a set of PEs, that is, software processors, custom hardware blocks, and memories. These components are selected by the system designer as part of the decision making. In particular, the designer selects the type and the number of PEs, CEs, and communication busses.

Architecture exploration consists of two tasks: PE allocation and partitioning. PE allocation defines the target architecture by selecting system components (software and hardware processors, memories) from the PE database. Partitioning then maps behaviors and variables to the allocated PEs and memories, respectively.

Following the design decisions of PE allocation and partitioning, the SCE architecture refinement tool `scar` inserts an additional layer of hierarchy representing the PEs into the model and groups behaviors and variables under these according to the partitioning. Next, it refines given complex channels into a client-server implementation using message-passing communication between the PEs and inserts necessary synchronization to properly preserve the original execution semantics. Finally, `scar` automatically generates the output architecture model [27].

3.2. Scheduling exploration

A key feature in the SCE design flow is the early evaluation of different scheduling strategies for software processors that are sequential and physically can only execute one task at a time. To evaluate different static and dynamic scheduling algorithms, such as round-robin or priority-based scheduling, we utilize a high-level RTOS model on each processor in the system [28]. Our abstract RTOS model is written on top of the SpecC language and does not require any specific language extensions. It supports all the key concepts found in modern RTOS, including task management, real-time scheduling, preemption, task synchronization, and interrupt handling.

After the designer chooses the desired scheduling strategy (e.g., round-robin, priority-based, or first-come-first-served), the SCE scheduling refinement tool `scos` automatically groups the given behaviors in the software PE into tasks and inserts the RTOS model with the user-defined scheduling strategy into the design model. `scar` then wraps all primitives and events that can trigger scheduling, such as task activation and termination, IPC synchronization and communication, and timing wait statements so that the inserted RTOS is called. It finally generates the refined model that can then be simulated for accurate observation and evaluation of dynamic scheduling behavior in the multitasking system. Since our abstract RTOS model requires only minimal overhead in simulation time, this approach enables early and rapid design space exploration.

3.3. Network exploration

Network exploration defines the system communication topology and maps the given communication channels onto a network of busses and communication elements (CEs), that is, bridges and transducers. For this, network refinement inserts the required CEs from the database into the model and implements the end-to-end communication over point-to-point links between PEs and CEs [29].

In the input architecture model, PEs communicate via abstract, typed end-to-end channels, and memory interfaces. During network exploration, the user allocates the actual communication media, bridges, and transducers for the system busses and CEs, respectively. Furthermore, the designer defines the connectivity of PE and CE ports to the busses,

and maps architecture-level end-to-end channels onto the allocated bus network.

Based on the network decisions by the designer, the SCE network refinement tool `scnr` inserts and implements the ISO/OSI presentation, network and transport layers, which implement data conversion, packeting, and routing; and acknowledgements, respectively. `scnr` then generates the new network model such that it reflects the selected network topology including typed end-to-end architecture level communication over untyped point-to-point links between the components in each network segment.

3.4. Communication synthesis

Next, the task of communication synthesis is to implement the point-to-point logical links between stations over the actual bus media, and to select and define the final pin- and bit-accurate parameters of the communication architecture under a set of constraints. Communication refinement then inserts protocols and bus-functional component descriptions from the bus and PE/CE databases, respectively, and generates a refined communication model that implements the communication links in each network segment over the actual, shared bus protocol and bus wires. In addition to this pin-accurate model (PAM), our communication refinement also generates a fast-simulating TLM of the system, which abstracts away the pin-level details of individual bus transactions [29].

In the input network model, communication in each network segment is described as a set of logical links. During communication synthesis, the designer (through the GUI, scripting or using synthesis plugins) defines the bus parameters, such as address and interrupt assignments, for each logical link over each bus. Based on these decisions, the SCE communication refinement tool `sccr` inserts low-level (transaction-level down to pin-accurate) models of busses and components from the databases, and generates a new communication model (PAM or TLM) of the design. In the output model, PE and CE components are refined to implement the lower communication layers (link, stream, media access, and protocol layer) for synchronization, addressing, and media accesses over each bus interface. On top of bus models from the bus database, the generated model hence implements all system communication down to the level of timing-accurate bus transactions (TLM), or cycle-accurate events for sampling and driving of the bus wires (PAM).

3.5. RTL synthesis

The task of RTL synthesis is to generate structural RTL from the behavioral description of the hardware components in the design. Although the designer can freely choose all behavioral synthesis parameters, including scheduling, allocation, and binding decisions, the SCE RTL synthesis tool `scrtl` supports automatic decision making through plugins. The designer can choose an algorithm to apply to all or only parts of their design. Critical parts of the design, on the other hand, can be manually preassigned or postoptimized [30].

Both designers and algorithms can rely on a set of estimates to aid them in the decision making. SCE includes RTL-specific profiling and analysis tools that provide feedback about a variety of metrics including delay, power, and variable lifetimes.

RTL synthesis in SCE takes full advantage of the designers' insight by allowing them to enter, modify, or override their decisions at will. On the other hand, tedious and error-prone tasks including code generation are automated.

3.6. Software synthesis

For implementing the software components in the system model, SCE relies on a layer-based modeling of the programmable processors and the software stack executing on them. Our embedded processor model supports task scheduling and interrupt handling.

Given scheduling priorities defined by the system designer, the SCE software synthesis tool sc2c automatically generates embedded software code for each processor from the system model [31]. More specifically, we generate efficient ANSI-C code from the SLDL code of the mapped application, and compile and link it against the selected RTOS. The resulting software binary can then be used for cycle-accurate instruction-set simulation within the system model, as well as for the final implementation.

4. EXPERIMENTS AND RESULTS

We have applied SCE to a large set of industrial-strength examples. In the following, we will first demonstrate the SCE design flow in detail as applied to a case study. Next, we summarize our experiences with different examples and show exploration results. Finally, we will present a set of verification experiments.

4.1. Modeling experiment

In order to demonstrate the overall SCE design flow, we have applied the flow to the example of a mobile phone baseband platform. The specification model of the system is shown in Figure 5. The design combines a JPEG encoder for processing of digital pictures taken by a camera and a voice encoder/decoder (vocoder) for speech processing based on the mobile phone GSM standard. Both JPEG and Vocoder processes are hierarchically composed of subbehaviors implementing the encoding and decoding algorithms in nested and pipelined loops and communicating through abstract message-passing channels. At the top level, a channel *Ctrl* between the two processes is used to send control messages from the JPEG encoder to the vocoder.

For the target platform (for space reasons, we do not show the platform model separately; the model is almost identical to Figure 6, with the exception that the OS layer and OS channel are omitted), we decide to use two software processors assisted by several hardware accelerators. For the JPEG encoder, we select a Motorola Coldfire processor for the main execution, assisted by a special IP component *DCT_IP* which performs the needed discrete cosine transformation

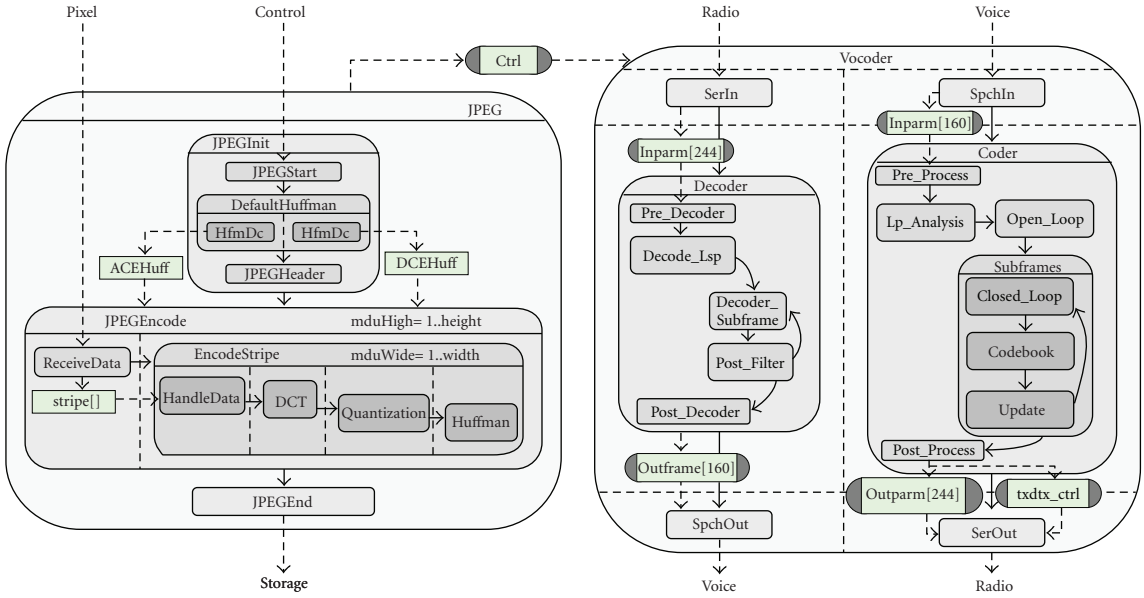


Figure 5: Baseband example: specification model.

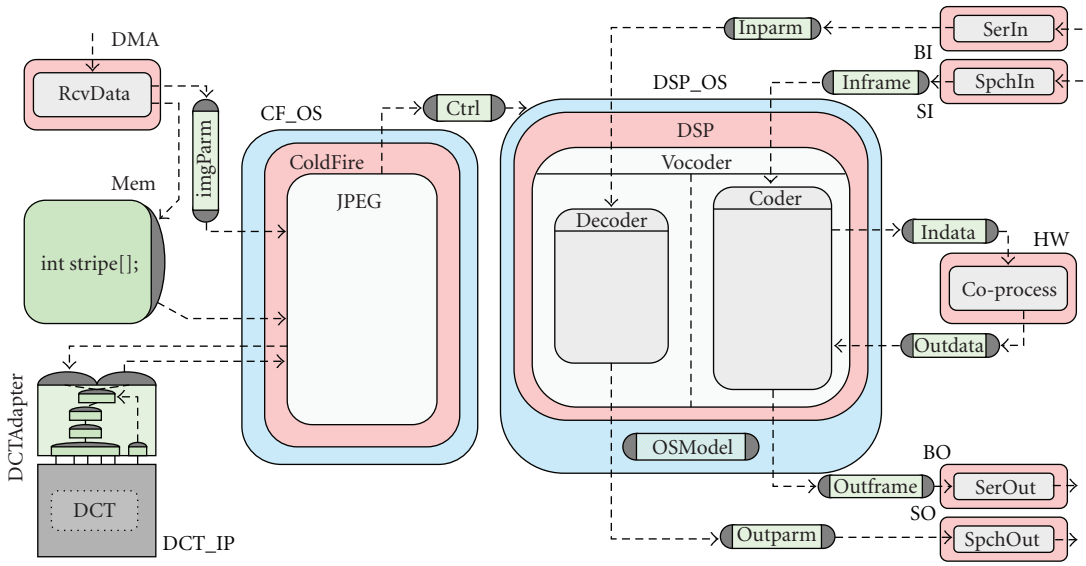


Figure 6: Baseband example: scheduled architecture model.

(DCT) in hardware. We also choose a direct memory access component *DMA* that receives pixel stripes from the camera and puts them into a shared memory *Mem*. On the other hand, we select a digital signal processor *DSP* to perform the majority of the voice encoding and decoding tasks. To reach the required performance, the *DSP* is assisted by four hardware blocks dedicated to input and output of the data streams, and one custom coprocessor in charge of the codebook search, the most time-critical function in the vocoder.

In the scheduled model obtained after architecture partitioning and scheduling (Figure 6), the *ColdFire* processor runs the JPEG encoder in software assisted by the hardware *DCT_IP*. Since this processor only executes this one task, no operating system is needed and the OS layer *CF_OS* is empty. On the other hand, the *DSP* performs two concurrent speech encoding and decoding tasks. These tasks are dynamically scheduled under the control of a priority-based operating system model that sits in an additional OS layer *DSP_OS* around the DSP. The encoder on the DSP is assisted by a custom hardware coprocessor (*HW*) for the codebook search. Furthermore, four custom hardware I/O processors perform buffering and framing of the vocoder speech and bit streams.

Table 1 summarizes the design decisions made for implementing the communication channels in the example. As a

Table 1: Communication design parameters for baseband example.

Channel	Network Routing	Addr.	Link Intr.	Medium
imgParm	linkDMA	0x00010000	int7	cfBus
stripe[]	Mem	0x0010*xxxx*	—	
hData dData	linkDCT	0x00010010	int1	
Ctrl	linkTx1	0x00010020	int2	
Ctrl	linkTx2	0xB000	intA	dspBus
inframe outparm	linkSI linkBO	0x800*x* 0x950*x*	intB	
indata outdata	linkHW	0xA000	intD	
inparm outframe	linkBI linkSO	0x850*x* 0x900*x*	intC	

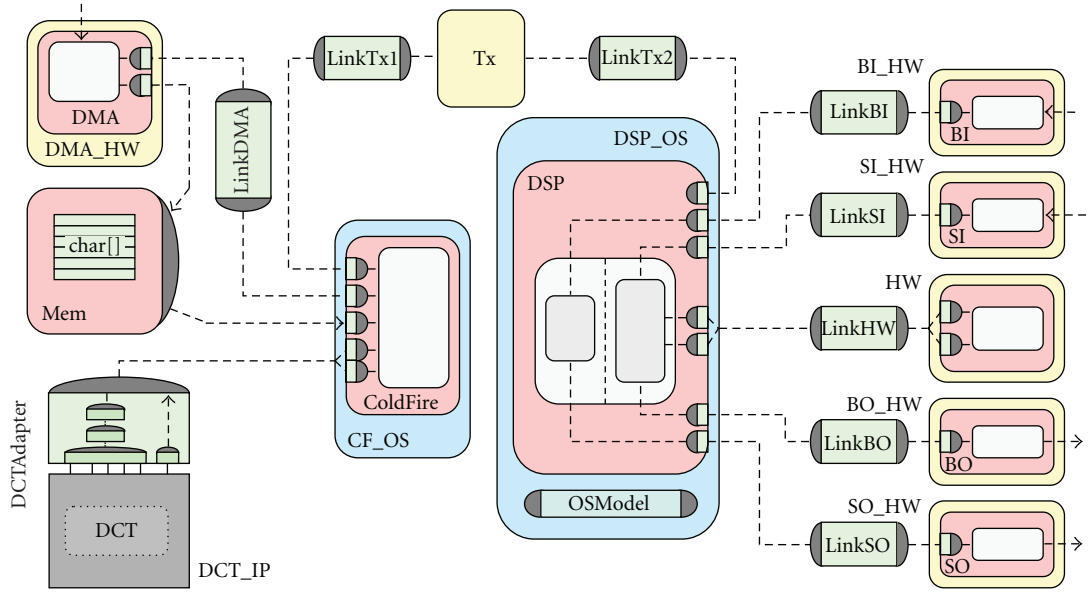


Figure 7: Baseband example: network model.

result of the network exploration, the network is partitioned into one segment per subsystem with a transducer *Tx* connecting the two segments (Figure 7). Individual point-to-point logical links connect each pair of stations in the resulting network model. Application channels are routed statically over these links where the *Ctrl* channel spanning the two subsystems is routed over two links via the intermediate transducer.

During communication synthesis, all links within each subsystem are implemented over a single shared medium. In both cases, the native ColdFire and DSP processor busses are selected as communication media. Within the segments, unique bus addresses and interrupts for synchronization are assigned to each link. On the ColdFire side, the memory is assigned a range of addresses with a base address plus offsets for each stored variable. On the DSP side, two of the four available interrupts are shared among the four I/O processors. In those cases, additional bus addresses for slave polling are assigned to each link (base address plus one). Finally, a bridge *DCT_Br* is inserted to translate between the *DCT_IP* and ColdFire bus protocols.

As a result, SCE communication synthesis generates two models, a fast-simulating TLM (Figure 8), and a pin-accurate model (PAM, Figure 9) for further implementation. In the TLM, link, stream, and media access layers are instantiated inside the OS and hardware layers of each station. Inside the

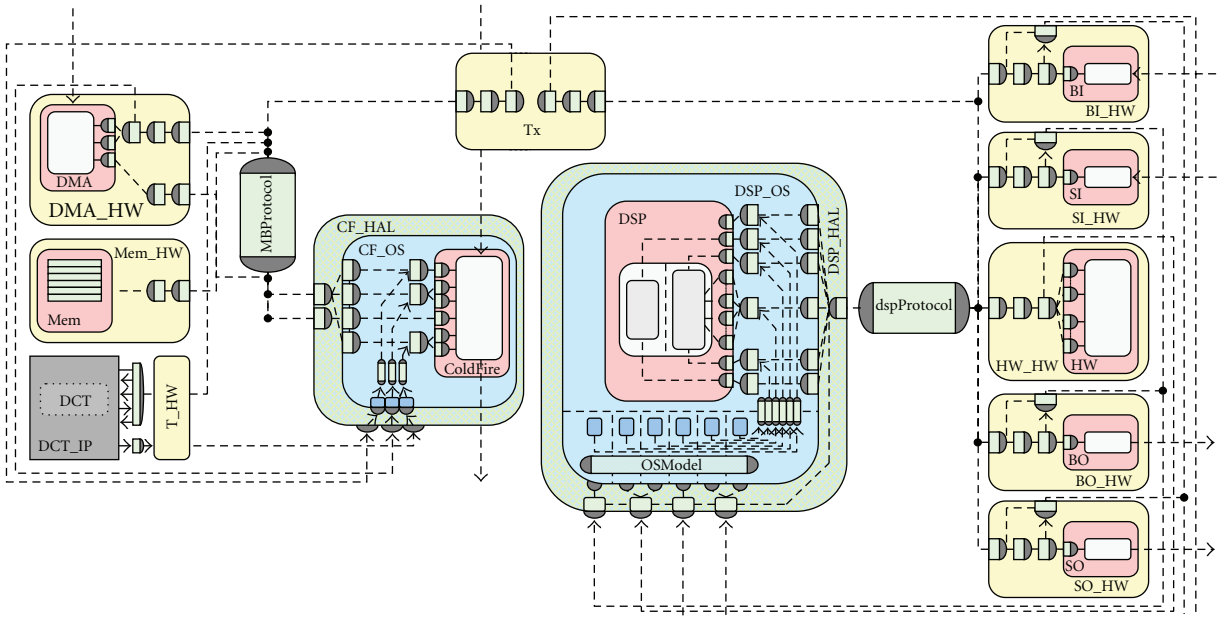


FIGURE 8: Baseband example: transaction-level model (TLM).

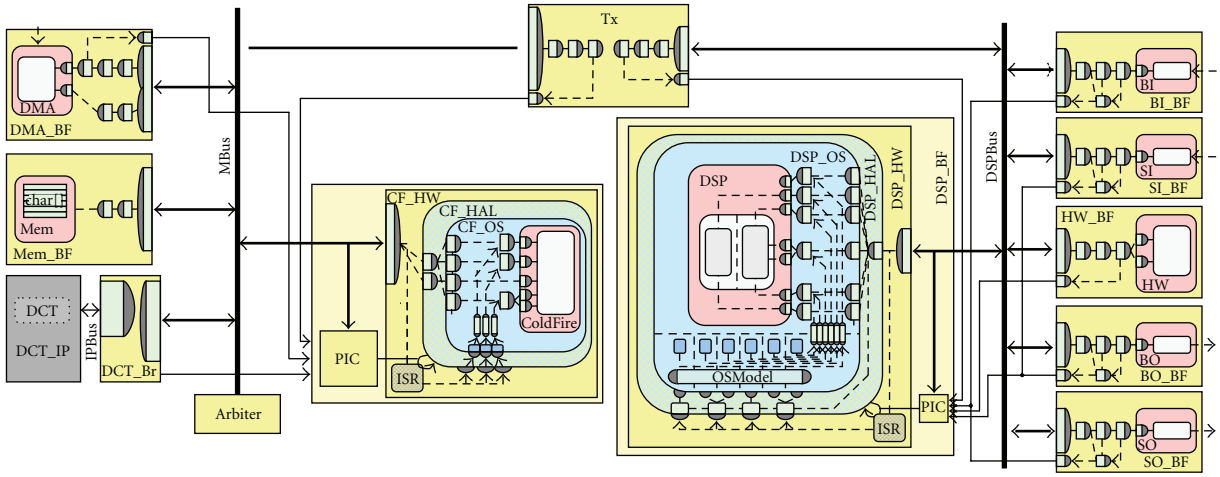


FIGURE 9: Baseband example: pin-accurate model (PAM).

processors, interrupt handlers that communicate with link layer adapters through semaphores are created. Interrupt service routines (*ISR*) together with models of programmable interrupt controllers (*PIC*) model the processor's interrupt behavior and invoke the corresponding handlers when triggered.

In the PAM, additionally the communication protocol layers are instantiated. Components are connected via pins and wires driven by the protocol layer adapters. On the ColdFire side, an additional arbiter component regulates bus accesses between the two masters, *DMA_BF* and *CF_BF*.

Table 2 summarizes the results for the example design. Using the refinement tools, models of the example design were automatically generated within seconds. A testbench common to all models was created which exercises the design by simultaneously encoding and decoding 163 frames of speech on the vocoder side while performing JPEG encoding of 30 pictures with 116×96 pixels. We created and refined both models of the whole system and models of each subsystem separately. Note that code sizes (lines of code, LOC) in each case include the testbenches. Since testbench code is shared, the size of the system model is less than the sum of the subsystem model sizes. All models were simulated on a 2.7 GHz Linux workstation using the QuickThreads version of the SpecC simulator.

Figure 10 plots simulation times on a logarithmic scale, that is, the graph shows that simulation times generally grow exponentially with each new model at the next lower level of abstraction. On the other hand, results of simulated overall frame transcoding (back-to-back encoding and decoding) and picture encoding delays in the vocoder and JPEG encoder, respectively, are shown in Figure 11. As can be seen,

Table 2: Modeling and simulation results for baseband example.

	ColdFire subsystem			DSP subsystem			System	
Model	LOC	Simul. time	JPEG delay	LOC	Simul. time	Vocoder delay	LOC	Simul. time
Specification	1,819	0.02 s	0.00 ms	9,736	1.31 s	0.00 ms	11,481	2.25 s
Architecture	2,779	0.03 s	9.66 ms	11,121	1.21 s	8.39 ms	13,866	2.56 s
Scheduled	3,098	0.02 s	22.63 ms	13,981	1.20 s	12.02 ms	17,020	2.00 s
Network	3,419	0.02 s	22.63 ms	14,319	1.22 s	12.02 ms	17,658	2.03 s
TLM	5,765	1.04 s	24.03 ms	15,668	27.4 s	13.00 ms	21,446	92.3 s
PAM	5,916	14.3 s	24.02 ms	15,746	34.8 s	13.00 ms	21,711	2,349 s
RTL-C	7,991	14.9 s	23.48 ms	23,661	147 s	12.88 ms	33,511	2,590 s

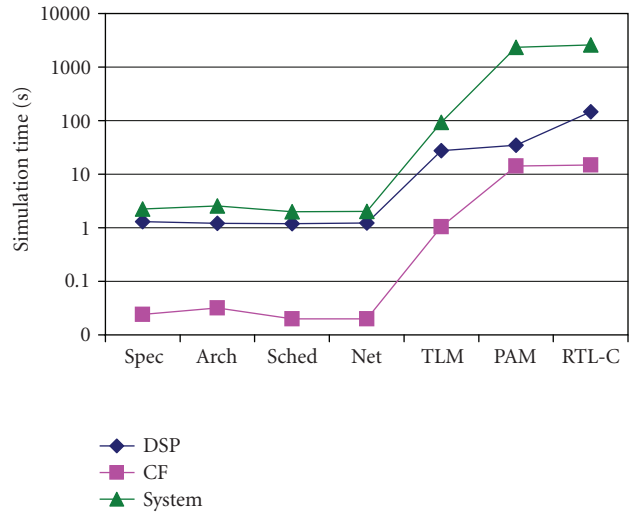


Figure 10: Simulation speeds for the baseband example.

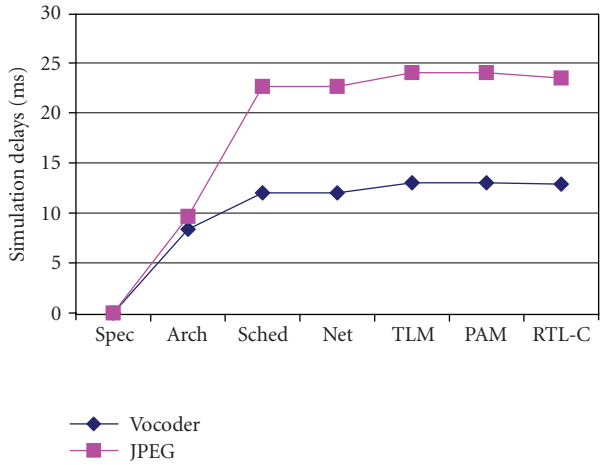


Figure 11: Simulated delays in the baseband example.

with each new model, measured delays linearly converge towards the final result.

Note that initial specification models are untimed and hence do not provide any delay measurements at all. Beginning with the architecture level, estimated execution

Table 3: Design examples and target architectures.

Examples		Buses (masters→slaves)
JPEG	A1	CF→HW
Vocoder	A1	DSP→HW
	A2	DSP→HW1,HW2
	A3	DSP→HW1,HW2,HW3
MP3float	A1	CF→HW1
	A2	CF→HW1,HW2,HW3
		HW1↔HW3
		HW2↔HW3
	A3	CF→HW1,HW2,HW3,HW4
		HW1↔HW3↔HW5
		HW2↔HW4↔HW5
MP3fix	A1	ARM→2 I/O
	A2	ARM→2 I/O, LDCT, RDCT
	A3	ARM→2 I/O, LDCT, RDCT
		LDCT↔I/O
		RDCT↔I/O
Baseband	A1	DSP→HW, 4 I/O, T
		CF,DMA→Mem,BR,T,DMA
		BR→DCTIP
Cellphone	A1	ARM→4 I/O, 2 DCT, T
		LDCT,RDCT→I/O
		DSP→HW, 4 I/O, T

delays are back-annotated into the computation blocks. As expected, scheduling has a large effect on simulation accuracy where abstract OS modeling enables evaluation of scheduling decisions at native simulation speeds (note that since the amount of simulated parallelism decreases, simulation is potentially even faster than at the specification level). Depending on the relation of communication versus computation, introducing bus models and communication delays at the transaction-level further increases accuracy, potentially at the cost of significantly longer simulation times. On the other hand, TLMs allow for accurate modeling of communication close or equivalent to pin-accurate models but at higher speed.

Table 4: Results for exploration experiments.

Examples		Model size (LOC)					Refinement time				
		Spec	Arch	Sched	Net	PAM	scar	scos	scnr	sccr	Total
JPEG	A1	1806	2409	2732	2780	4642	0.27 s	0.37 s	0.16 s	0.21 s	1.01 s
Vocoder	A1	7385	8449	9594	9775	10679	2.29 s	1.30 s	0.62 s	0.56 s	4.77 s
	A2		8508	9632	9913	10989	2.41 s	1.36 s	0.75 s	0.69 s	5.21 s
	A3		8535	9659	9949	11041	2.64 s	1.69 s	0.79 s	0.64 s	5.76 s
MP3float	A1	6900	6963	28190	28204	29807	0.82 s	3.24 s	0.90 s	0.90 s	5.86 s
	A2		7181	28275	28633	31172	0.93 s	2.66 s	1.11 s	1.48 s	6.18 s
	A3		11069	28736	30202	32795	4.66 s	4.05 s	7.10 s	1.88 s	17.69 s
MP3fix	A1	13363	13724	17131	17270	21593	0.95 s	1.37 s	0.58 s	0.95 s	3.85 s
	A2		16040	18300	18564	23228	3.28 s	1.68 s	0.85 s	1.20 s	7.01 s
	A3		16023	18748	19079	24471	2.72 s	1.76 s	1.97 s	0.95 s	7.40 s
Baseband	A1	11481	13866	17020	17658	21711	4.27 s	2.46 s	1.24 s	1.02 s	8.99 s
Cellphone	A1	16441	18653	21936	22570	30072	3.86 s	3.10 s	1.31 s	1.22 s	9.49 s

Table 5: Results for equivalence verification.

Examples	Refinement	Model 1			Model 2			No. of transformations	Verification time
		Type	No. of nodes	No. of edges	Type	No. of nodes	No. of edges		
JPEG	Architecture	spec.	148	219	arch.	180	257	1602	1.6 s
	Scheduling	arch.	180	257	sched.	180	287	2740	2.1 s
	Network	sched.	180	287	net.	201	253	2852	2.1 s
Vocoder	Architecture	spec.	436	761	arch.	528	882	6131	3.3 s
	Scheduling	arch.	528	882	sched.	528	881	7065	3.7 s
	Network	sched.	528	881	net.	569	933	7229	3.8 s

Our results show that with increasing implementation detail at lower levels of abstraction, accuracy (as measured by the simulated delays) improves linearly while model complexities (as measured by code sizes and simulation times) grow exponentially. All in all, our results support the choice of intermediate models in the design flow that allows for fast validation of critical design aspects at early stages of the design process.

4.2. Exploration experiments

In order to demonstrate our approach in terms of design space exploration for a wide variety of designs, we applied SCE to the design of six industrial-strength examples: standalone versions of the JPEG encoder (*JPEG*) and the GSM voice codec (*Vocoder*), floating- and fixed-point versions of an MP3 decoder (*MP3float* and *MP3fix*), the previously introduced baseband example (*Baseband*), and a *Cellphone* example combining the JPEG encoder, the MP3 decoder, and the GSM vocoder in a platform mimicking the one used in the RAZR cellphone. For each example, we generated different architectures using Motorola DSP56600 (*DSP*), Motorola ColdFire (*CF*), and ARM7TDMI (*ARM*) processors together with custom hardware coprocessors (*HW*, *DCT*) and *I/O* units. We used various communication architectures with *DSP*, *CF*, *ARM* (AMBA AHB), and simple handshake busses.

Table 3 summarizes the features and parameters of the different design examples we tested. For each example, the target architectures are specified as a list of masters plus slaves for each bus in the system where the bus type is implicitly determined to be the protocol of the primary master on the bus. For example, in the case of the *MP3float* design, the ColdFire processor communicates with dedicated hardware units over its *CF* bus whereas the *HW* units communicate with each other through separate handshake busses. For simplicity, routing, address, and interrupt assignment decisions are not shown in this table.

Table 4 shows the results of exploration of the design space for the different examples. Overall model complexities are given in terms of code size using lines of code (LOC) as a metric. Results show significant differences in complexity between input and generated output models due to extra implementation detail added between abstraction levels.

Note that manual refinement would require tremendous effort (in the order of days). Automatic refinement, on the other hand, completes in the order of seconds. Our results therefore show that a significant productivity gain can be achieved using SCE with automatic model refinement.

4.3. Verification experiments

We implemented the SCE equivalence verification tool *scver* to verify the refinements above network level. Since the

lowest abstraction level of communication in model algebra is the channel, models below network level in the SCE flow could not be directly translated into model algebraic representation.

The results for verification of architecture, scheduling, and network refinements are presented in Table 5. We used two benchmarks, namely, the JPEG encoder and Vocoder as shown in column 1. The model algebraic representation was stored in a graph data structure, with nodes being the objects and edges being the composition rules. Column 5 shows the total transformations applied to derive model 1 from model 2 using the transformation rules of model algebra. As we can see, since the order of transformation is decided, it only took a few seconds to apply them even for representations with hundreds of nodes and edges. The verification time also includes the time it took to parse the SpecC models into model algebraic representation and to perform isomorphism checking between the derived and original model graphs.

The results demonstrate that the SCE tool flow based on well-defined model abstractions and semantics enables fast equivalence verification.

5. SUMMARY AND CONCLUSION

In this work, we have presented SCE, a comprehensive system design framework based on the SpecC language. SCE supports a wide range of heterogeneous target platforms consisting of custom hardware components, embedded software processors, dedicated IP blocks, and complex communication bus architectures.

The SCE design flow is based on a series of automated model refinement steps where the system designer makes the decisions and SCE quickly provides estimation feedback, generates new models automatically, and validates them through simulation and formal verification. The effective design automation tools integrated in SCE allow rapid and extensive design space exploration. The fast exploration capabilities, in turn, enable the designer to optimize the system architecture, the scheduling policies, the communication network, and the hardware and software components, so that an optimal implementation is reached quickly.

We have demonstrated the benefits of SCE by use of six industrial-size examples with varying target architectures, which have been designed and verified top-to-bottom. Compared to manual coding and model refinement, SCE achieves productivity gains by orders of magnitude.

SCE has been successfully transferred to and applied in industrial settings. SER, a commercial derivative of SCE, has been developed and integrated into ELEGANT, an environment for electronic system-level (ESL) design of space and satellite electronics that was commissioned by the Japanese Aerospace Exploration Agency (JAXA). ELEGANT and SER have been succesfully delivered to JAXA's suppliers and are currently being introduced into the general market [32].

ACKNOWLEDGMENTS

The authors would like to thank all members of the CECS SpecC group who have contributed to SCE over the years. Special thanks go to David Berner, Pramod Chandraiah, Quoc-Viet Dang, Alexander Gluhak, Eric Johnson, Raphael Lopez, Gunar Schirner, Ines Viskic, Shuqing Zhao, and Jianwen Zhu.

REFERENCES

[1] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Design Methodology*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[2] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, Upper Saddle River, NJ, USA, 1994.

[3] P. Coste, F. Hessel, Ph. Le Marrec, et al., "Multilanguage design of heterogeneous systems," in *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES '99)*, pp. 54–58, Rome, Italy, May 1999.

[4] P. Gerin, S. Yoo, G. Nicolescu, and A. A. Jerraya, "Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '01)*, pp. 63–68, Yokohama, Japan, January-February 2001.

[5] *ModelSim SE User's Manual*, Mentor Graphics Corp.

[6] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: a framework for simulating and prototyping heterogeneous systems," *International Journal of Computer Simulation*, vol. 4, no. 2, pp. 155–182, 1994.

[7] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[8] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts, *Handel-C language reference guide*, Oxford University Computing Laboratory, Oxford, UK, August 1996.

[9] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*, Springer, New York, NY, USA, 2005.

[10] A. Österling, T. Brenner, R. Ernst, D. Herrmann, T. Scholz, and W. Ye, "The COSYMA system," in *Hardware/Software Co-Design: Principles and Practice*, J. Staunstrup and W. Wolf, Eds., Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[11] C. A. Valderrama, M. Romdhani, J.-M. Daveau, G. F. Marchioro, A. Changuel, and A. A. Jerraya, "Cosmos: a transformational co-design tool for multiprocessor architectures," in *Hardware/Software Co-Design: Principles and Practice*, J. Staunstrup and W. Wolf, Eds., Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[12] F. Balarin, M. Chiodo, P. Giusto, et al., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[13] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens, "Hardware/software partitioning for embedded systems in OCAPI-xl," in *Proceedings of the International Symposium on Hardware-Software Codesign (CODES '01)*, Copenhagen, Denmark, April 2001.

[14] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens, "A programming environment for the design of complex high speed ASICs," in *Proceedings of the 35th Annual Conference on Design Automation (DAC '98)*, pp. 315–320, San Francisco, Calif, USA, June 1998.

[15] K. Grüttner, F. Oppenheimer, W. Nebel, A.-M. Fouilliart, and F. Colas-Bigey, "SystemC-based modelling, seamless refinement, and synthesis of a JPEG 2000 decoder," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE '08)*, pp. 128–133, Munich, Germany, March 2008.

[16] W. O. Cesário, D. Lyonnard, G. Nicolescu, et al., "Multiprocessor SoC platforms: a component-based design approach," *IEEE Design and Test of Computers*, vol. 19, no. 6, pp. 52–63, 2002.

[17] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," in *Proceedings of the 38th Annual Conference on Design Automation (DAC '01)*, pp. 518–523, Las Vegas, Nev, USA, June 2001.

[18] K. van Rompaey, I. Bolsens, H. De Man, and D. Verkest, "CoWare—a design environment for heterogeneous hardware/software systems," in *Proceedings of the European Design Automation Conference (EURO-DAC '96)*, pp. 252–257, Geneva, Switzerland, September 1996.

[19] W. Klingauf, H. Gädke, and R. Günzel, "TRAIN: a virtual transaction layer architecture for TLM-based HW/SW codesign of synthesizable MPSoC," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE '06)*, vol. 1, Munich, Germany, March 2006.

[20] T. Kempf, M. Doerper, R. Leupers, et al., "A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE '05)*, vol. 2, pp. 876–881, Munich, Germany, March 2005.

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

[22] A. L. Sangiovanni-Vincentelli, "Quo vadis SLD: reasoning about trends and challenges of system-level design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[23] S. Abdi, J. Peng, H. Yu, et al., "System-on-chip environment (SCE version 2.2.0 beta): tutorial," Tech. Rep. CECS-TR-03-41, Center for Embedded Computer Systems, University of California, Irvine, Calif, USA, July 2003.

[24] L. Cai, A. Gerstlauer, and D. Gajski, "Retargetable profiling for rapid, early system-level design space exploration," in *Proceedings of the 41st Annual Conference on Design Automation (DAC '04)*, pp. 281–286, San Diego, Calif, USA, June 2004.

[25] S. Abdi and D. Gajski, "Verification of system level model transformations," *International Journal of Parallel Programming*, vol. 34, no. 1, pp. 29–59, 2006.

[26] A. Gerstlauer, L. Cai, D. Shin, H. Yu, J. Peng, and R. Dömer, *SCE database reference manual, version 2.2.0 beta*, Center for Embedded Computer Systems, University of California, Irvine, Calif, USA, July 2003.

[27] J. Peng and D. Gajski, "Optimal message-passing for data coherency in distributed architecture," in *Proceedings of the 15th International Symposium on System Synthesis*, pp. 20–25, Kyoto, Japan, October 2002.

[28] A. Gerstlauer, H. Yu, and D. D. Gajski, "Rtos modeling for system level design," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE '03)*, Munich, Germany, March 2003.

[29] A. Gerstlauer, D. Shin, J. Peng, R. Dömer, and D. D. Gajski, "Automatic layer-based generation of system-on-chip bus communication models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 9, pp. 1676–1687, 2007.

[30] D. Shin, A. Gerstlauer, R. Dömer, and D. D. Gajski, "An interactive design environment for C-based high-level synthesis of RTL processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 4, pp. 466–475, 2008.

[31] H. Yu, R. Dömer, and D. Gajski, "Embedded software generation from system level design languages," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, pp. 463–468, Yokohama, Japan, January 2004.

[32] *CECS eNews Volume 7, Issue 3*, Center for Embedded Computer Systems, University of California, Irvine, Calif, USA, July 2007, http://www.cecs.uci.edu/enews/CECSeNewsJul07.pdf.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 134798, 11 pages
doi:10.1155/2008/134798

Research Article

Modelling Field Bus Communications in Mixed-Signal Embedded Systems

Mohamad Alassir, Julien Denoulet, Olivier Romain, Abraham Suissa, and Patrick Garda

Université Pierre et Marie Curie – Paris 6, EA 2385, BC 252-4 Place Jussieu, 75252 Paris Cedex 05, France

Correspondence should be addressed to Julien Denoulet, julien.denoulet@upmc.fr

Received 1 October 2007; Revised 29 February 2008; Accepted 10 June 2008

Recommended by Christoph Grimm

We present a modelling platform using the SystemC-AMS language to simulate field bus communications for embedded systems. Our platform includes the model of an I/O controller IP (in this specific case an I^2C controller) that interfaces a master microprocessor with its peripherals on the field bus. Our platform shows the execution of the embedded software and its analog response on the lines of the bus. Moreover, it also takes into account the influence of the circuits's I/O by including their IBIS models in the SystemC-AMS description, as well as the bus lines imperfections. Finally, we present simulation results to validate our platform and measure the overhead introduced by SystemC-AMS over a pure digital SystemC simulation.

1. INTRODUCTION

Nowadays, the design of embedded systems is a key issue for the electronics industry [1]. Many ongoing research projects address this issue by increasing the design abstraction level from the circuit to the system level [2]. However, most embedded systems gather analog and digital electronics with embedded processor cores running large amounts of embedded software.

A number of issues are unfortunately discovered after the conception of embedded systems prototypes. These issues include for example coupling between analog and digital functions or unexpected hazard resulting from unexpected low level effects. Of course, discovering such issues at the prototyping step and revising the design and its implementation to get rid of them is time and money consuming. Addressing these issues at the design time requires a mixed-signal model gathering digital and analog functions [3]. A number of approaches exist to provide these mixed-signal models. They are considering different levels of abstraction and different models of computations. These levels and models are simulated with different tools and cosimulation is used to cover the relations between them.

To be more specific, we observe that neither VHDL-AMS nor SystemC are perfectly well suited to model the architecture of these embedded systems. On the one hand, VHDL-AMS is perfect to model analog and digital electronics, but the simulation speed of processor cores is too slow in VHDL for the simulation of embedded software [4]. On the other hand, SystemC is perfect to model digital electronics and processor cores executing embedded software, but it cannot model analog electronics [5].

These observations led to the proposal of SystemC-AMS, for which beta releases of the language and the simulation engine are now available [6]. SystemC-AMS provides, in addition to the SystemC discrete-event simulation kernel, continuous-time models of computations for conservative and nonconservative system modelling [7–9]. Moreover, a specific AMS working group was created by OSCI [10] in 2006 to specify SystemC-AMS [11].

We want to investigate the potential of SystemC-AMS to provide a unified framework for the design of embedded systems. It could give the advantages of multiple levels and multiple computational models in a single language based on an open standard. Of course, as SystemC-AMS is still under specification, it is unclear as of today if this is the most valid approach, or if it provides significant gains over other approaches, but the objective of our work presented in this paper is to contribute to this specification and to provide the

AMSWG with some applications of mixed-signal embedded systems.

Considering the long-term challenge we are addressing and the need for specific performances data to provide a significant insight, we chose a pragmatic step-by-step research strategy, and we begun with a rather simple case of mixed-signal embedded system.

In many different applications, (automotive, industrial control, etc.), embedded systems are typically formed by a set of nodes connected through field busses such as I^2C or CAN. The nodes are typically mixed analog and digital SOCs, including one (or several in the near future) processor core(s) and an array of peripheral interfaces. In its most usual form, the node is simply a microcontroller, connected to various sensors or actuators.

In this paper, we show how to model the field bus communications between the nodes of an embedded system. Our methodology is based on three main parts: firstly, a generic architecture of an I/O controller realizing the interface between the processor cores and an I^2C field bus; secondly, a generic architecture for the analog part of this interface, and a software tool to get the parameters of this interface from industrial IBIS files; thirdly, a generic model of the bus lines. Hence, our method is generic and it can be applied to any field bus. This paper is based on our previous work on the design and modelling of an I^2C bus controller in VHDL [12] and SystemC-AMS [13, 14]. It is an extended revision of the (unpublished) paper presented in [15].

For didactical purposes, the paper is organised around a platform presented in Figure 1. It represents a common situation, where a microcontroller is connected to a System-on-Chip through a field bus. Our model includes discrete-event models for the digital cores, implemented in SystemC, and continuous-time models for the analog interfaces and for the bus lines, implemented in SystemC-AMS. Whereas this methodology is generic, we will present it in the specific case of the I^2C and CAN busses.

The paper is divided into three main sections. Section 2 describes the architecture of the digital I/O controller. Section 3 presents the architecture of the analog I/O, the method to include data from an IBIS model into an equivalent SystemC-AMS description of a circuit's I/O buffers, and the electrical model of the bus lines. Section 4 gives several simulation results of field bus communication on the platform and estimates the overhead of the analog parts of the models compared to a totally digital simulation.

2. DIGITAL MODELS

The modelling platform discussed below is presented in Figure 2. It is centered on a field bus that connects four nodes: among them two can be masters on the bus, while the remaining two can only be slaves. The two masters are a 8051 microcontroller in association with a bus controller and a MIPS-based system on Chip. The two slaves are memory devices which will be accessed in the course of simulation by the two masters. Each node is modeled with a mixed SystemC/SystemC-AMS description, where a device's digital core is modeled in SystemC and its analog interface

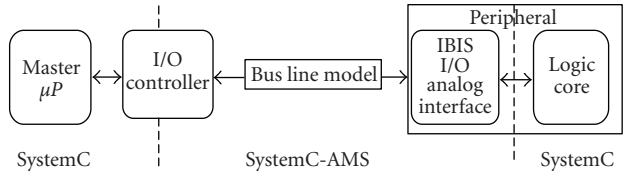


Figure 1: General structure of the modelling platform.

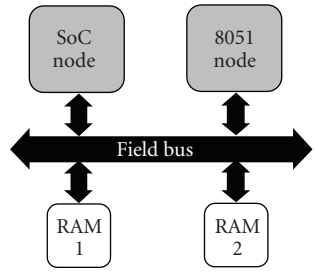


Figure 2: Field-bus based modelling platform.

is modeled in SystemC-AMS. This section introduces the digital architecture of the platform components, especially focusing on the bus controller.

2.1. Bus controller

We first decided to model our platform around an I^2C bus, mainly because this standard provided a simple protocol and an easy-to-implement mixed-signal interface. However, it seemed appropriate to work on a generic structure that could be applied to different protocols with only a minimum of changes. Thus, this subsection presents our controller architecture applied to the case of an I^2C bus.

2.1.1. I^2C protocol

Historically, the first I^2C (Inter-Integrated Circuit) controllers and the I^2C protocol were designed by Philips at the beginning of the eighties for television applications [16]. The I^2C bus was invented to provide communication on a two-wire bidirectional bus—a serial data (SDA) and clock (SCL)—between a small number of devices (sensors, LCD, microcontroller, etc.). Transfer frequency is up to 100 kbits/s for standard mode, and up to 3.4 Mbits/s in high-speed mode.

Data frame for the standard mode is made of a start bit, a 7 bits address, a Read/Write bit, an acknowledge bit (ACK), and a sequence of data bytes. Each data byte is followed by an acknowledge bit issued by the target device. A stop bit finalizes the transmission (Figure 3). Each bit is transmitted on SDA in conjunction with the SCL clock.

(i) Transfers are initiated by a START condition. It happens when a falling transition occurs on the SDA line while SCL is high.

(ii) Transfers end with a STOP condition. It happens when a rising transition occurs on the SDA line while SCL is high (Figure 4).

Figure 3: I²C data frame.

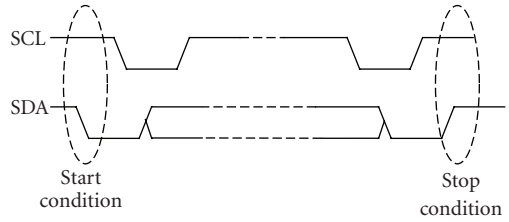


Figure 4: Start and stop condition.

(iii) Data is considered valid during the high state of SCL. Therefore, SDA signal must remain stable during this half period (Figure 5).

(iv) I²C allows multimaster communications and features an arbitration management protocol for such transmissions. Arbitration takes place on the SDA line, while the SCL line is at the high level. Bus control is given to the master which transmits a low level while the others transmit a high level. A master which loses arbitration switches to high impedance its data output stage (Figure 6).

As a specification for our I²C controller model, we chose the PCF8584 designed by Philips/NXP [17].

2.1.2. I²C controller digital block architecture

The architecture of the digital block is divided into three blocks (Figure 7).

(i) A microprocessor interface handles communication with the microprocessor core. It is built around a FIFO that stores the successive requests coming from the microprocessor bus (8051 bus, VCI, AMBA, etc.). When the controller has finished a transaction on the bus, and if a request is stored in the FIFO, the FIFO is read and the interface extracts the information needed by the sequencer (type of operation, address, data) to perform the new communication. The interface also includes an interrupt line connected to the microprocessor core so that it can read a data received from the I²C bus.

(ii) The core of the controller is a sequencer which translates the request from the master into a detailed sequence respecting the I²C protocol (frame generation, byte transmission, or reception, etc.).

(iii) Finally, a signal generator module manages or drives the SCL and SDA bus lines (Figure 8). This block manages the acknowledge generation/detection depending on the operating mode (transmitter or receiver). A shift register either serializes data to be sent to the bus line in transmitting mode or collects information from the bus in reception mode. It also sets the transmission frequency by dividing the system clock with a user-defined constant. The SDA and SCL signals from this block will be interfaced with the analog model presented in Section 3.

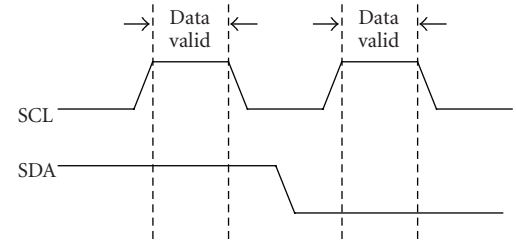


Figure 5: Data validity period.

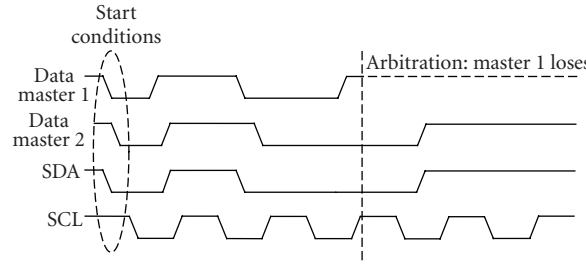


Figure 6: Arbitration procedure with two masters.

2.2. Other devices

To develop the SoC node of the platform, we used a virtual prototyping platform for system-on-chips called SoCLib [18]. SoCLib provides a number of IP SystemC models (including processor cores, VCI interconnect, memory devices, I/O controllers, timers, etc.), to easily simulate an application on a user-assembled SoC or MPSoC architecture built around a virtual component interface (VCI) bus [19].

We added our controller model to the SoCLib library and used it to connect an MIPS processor core to an external field bus. To do so, a SystemC wrapper has been developed to interface our controller to the VCI bus provided by SoCLib. The platform also includes a RAM component used by the MIPS to store its code and data and a TTY terminal for debug purposes. A SoCLib-generated platform synopsis can be found in Figure 9.

We will not detail the remaining models, the I²C RAM, and the 8051 microcontroller [20]. The first of this model is a basic description of a memory component while the second one is a standard description of this well-known circuit.

3. ANALOG MODELLING

We introduce in this section the analog modules of our platform. All these devices were modeled in SystemC-AMS v.0.15. First, we show the analog interface of the bus controller, designed according to the requirements of the I²C protocol. Then, for the slave devices, we present an interface retrieved from an IBIS model of a standard component and show how such IBIS models can be automatically translated into a SystemC-AMS description. Finally, we focus on the transmission lines, introducing a model that takes into account the wires' physical imperfections as well as the mutual influence of adjacent transmission lines.

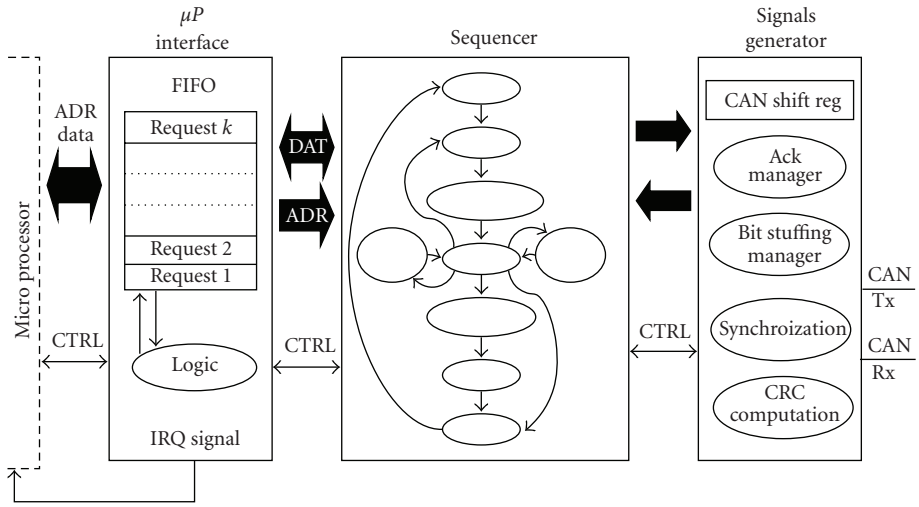


Figure 7: Controller digital block architecture.

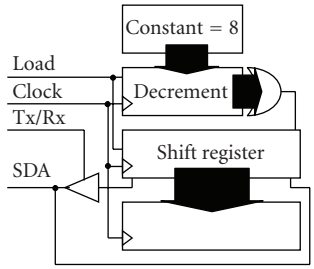


Figure 8: SDA line manager.

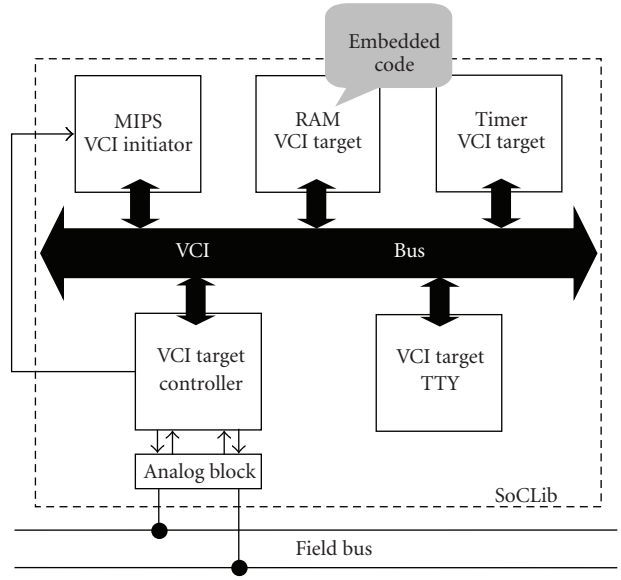


Figure 9: SoCLib simulation platform.

3.1. Bus controller

Specifications of the I²C protocol indicate that the devices must have open-drain or open-collector outputs (depending on technology) in order to perform a wired-AND function to manage multimaster mode. Both lines also feature a pullup resistor to VCC (Figure 10(a)).

We modeled this analog interface with the SystemC-AMS 0.15 library, by instantiating linear elements (resistors, capacitors, voltages sources, etc.). Figure 10(b) represents the interface model for the SDA line. It translates the logic levels sent by the digital block to voltages across the bus lines. The SCL model is similar.

In this model, the transistor is represented with an interrupter and a resistor as no transistor model is available in SystemC-AMS 0.15. A 20 pF capacitor is used to manage the rising and falling times of the SDA signal. Finally, a pullup resistor sets the line at a high state when no command is applied on the bus. To read a value coming from the bus, a threshold detection is applied to the SDA line and it provides a logical value to the digital block.

Algorithm 1 gives a SystemC-AMS sample code of the analog interface for either line of the bus. The electrical parameters of the devices (including the output transistor) were chosen to be in accordance with specifications found in NXP's PCF8584 controller datasheet [17].

3.2. IBIS to SystemC-AMS conversion

In order to accurately model the analog behaviour of the components connected on the bus, and integrate real parameters of industrial components to our SystemC-AMS models, we developed an IBIS-to-SystemC-AMS conversion tool. It parses a standard IBIS source file to produce a SystemC-AMS module we can directly instantiate in our platform. This section briefly presents the IBIS standard and then shows how these models can be used in our simulation platform.

3.2.1. IBIS specifications

The input/output buffer information specification (IBIS) [21] standard was originally introduced to give a behavioural

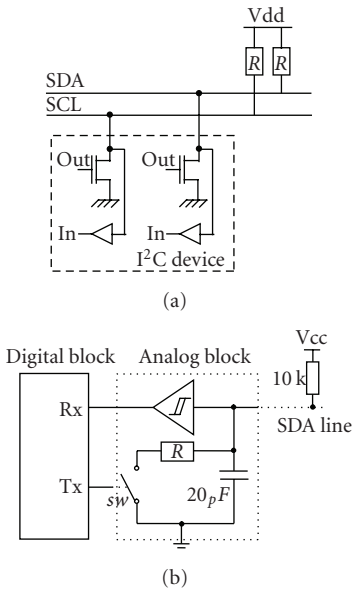


Figure 10: (a) I²C electrical scheme (b) SystemC-AMS model.

description of a circuit's inputs/outputs. This behaviour is based on voltage/current curves obtained from measurements or full simulation. It especially takes into account the influence of the chip's package on the I/O signal form.

An IBIS model is presented as a text file which includes for each input or output pad an R, L, C equivalent circuit modelling the influence of the package as well as another circuit representing the response of the pad to a rising or falling edge transition.

Figure 11 presents a model of an output buffer (input buffers are modeled in a similar way). The voltage/current curves included in the IBIS file give the response of the CMOS inverter to a logic transition. However, the IBIS standard specifies that this response is equivalent to another R, L, C circuit. As a result, a SystemC-AMS module of the electrical scheme presented in Figure 12 can accurately model the behaviour of the output buffer. The figure also shows how the module can be interfaced with a SystemC description of the logic core of the component.

3.2.2. IBIS-to-SystemC-AMS conversion

To add IBIS models in our SystemC-AMS simulation platform, we developed an IBIS file analyzer program which parses the IBIS file to extract the I/O circuits useful information and then automatically creates a SystemC-AMS module using these parameters.

An IBIS file is made of several sections that first present general information about the chip, display the R, L, C values of the circuit's package, enumerate the different I/O pins, and specify their type (input, output, GND, etc.), then they describe the behaviour of each I/O type found in the chip. A sample of an IBIS file is presented in Algorithm 2. Each section is specifically introduced by a keyword (such as [Package] or [Pin] in the file sample). The aim of our tool

```
SCA_SDF_MODULE(ADC){    // Threshold Detector
   sca_sdf_in<double> in;
   sc_out<bool> out;
   ...
   void init(){}
   void sig_proc(){ if(in.read() > 2.5) out = 1;
                    else out = 0;}
   ...
   SCA_CTOR(ADC){}
};
   ...
SCA_SDF_MODULE(data_conv){
   sca_sdf_in<double> portin;
   sc_out<double> portout;
   ...
   double portout_;
   void init(){ portout = 1.0e3;}
   void sig_proc(){ portout_ = portin.read();
                    portout = portout_ * 3.0e2;}
   ...
   SCA_CTOR(data_conv){}
};
   ...
SC_MODULE(SDA_Mgr){
   sc_in<bool> sda_in;     // From digital block
   sc_out<bool> sda_out;   // To digital block
   sca_sdf_signal<double> sda_sdf; // To I2C Bus
   ...
   sc_signal<double> r_value;
   sc_signal<bool> sig_;
   sca_elec_port out;    // Output port
   sca_elec_port gnd;    // Ground port
   sca_elec_node y;      // Internal node
   ...
   sca_sc2r *r1;                sca_c *c1;
   sca_rswitch *sw1;            sca_vd2sdf *conv1;
   data_conv *data_conv1;       can *can1;
   ...
   SC_CTOR(SDA_Mgr){
   ...
      ADC1 = new ADC("ADC1");    // Threshold
             ADC1->in(sda_sdf); ADC1->out(sda_out);
   ...
         sw1 = new sca_rswitch("sw1");    // Switch
         sw1->off_val = true; sw1->p(y);
         sw1->n(gnd); sw1->ctrl(sda_in);
   ...
      r1 = new sca_sc2r("r1");    // Resistor
           r1->ctrl(r_value); r1->p(out); r1->n(y);
   ...
      data_conv1 = new data_conv("data_conv1");
         data_conv1->portin(sda_sdf);
         data_conv1->portout(r_value);
   ...
      c1 = new sca_c("c1");    // Capacitor
         c1->value = 20.0e − 12;
         c1->p(out); c1->n(gnd);
   ...
      conv1 = new sca_vd2sdf("conv1"); // Output voltage
         conv1->p(out); conv1->n(gnd);
         conv1->sdf_voltage(sda_sdf);
      }
};
```

Algorithm 1: I²C controller analog block code sample.

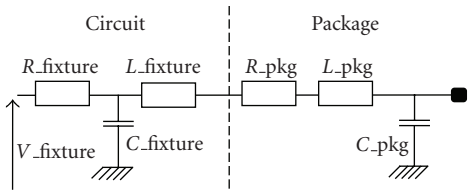


Figure 11: IBIS output model.

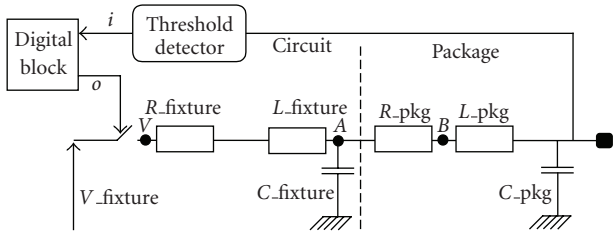


Figure 12: IBIS equivalent circuit to model in SystemC-AMS.

```
[Package]
| variable      typ          min          max
R_pkg           0.20         100.00 m     0.40
L_pkg           7.00 nH      4.10 nH      10.00 nH
C_pkg           1.50 pF      1.00 pF      3.00 pF
|
|**********Pin Defintions**********
[Pin]           signal_name      model_name
|
1               A0               InputModel1
2               A1               InputModel1
3               A2               InputModel1
4               GND              GND
5               SDA              IOModel
6               SCL              InputModel2
7               WP               InputModel3
8               VCC              POWER
```

Algorithm 2: IBIS text file sample.

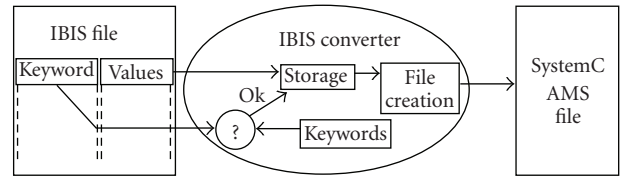


Figure 13: IBIS-to-SystemC-AMS converter tool synopsis.

is to scan the IBIS file, look for those keywords, and extract useful information, in particular the number of each I/O type as well as each type's electrical characteristics, that is, the value of the R, L, C elements.

The synopsis of our IBIS-to-SystemC-AMS converter is presented in Figure 13. When running, it performs a scanning phase followed by a generation phase.

When it parses the IBIS file, the program first counts the number of different I/O models for the circuit as well as their number of occurrences. Then, for each I/O model, it extracts and stores the value of the R, L, C elements. To do so, the program compares the first word of each line of the IBIS file with a set of predefined keywords (e.g., "R_pkg" for the package resistance). If it matches with one of the keywords, it saves the value of the element (which, in an IBIS file, is usually written next to the keyword).

As an illustration, in the IBIS text file sample (Algorithm 2), the program will count 8 I/O, among them three identical (A0, A1, A2). It will also notice the R_pkg, L_pkg and C_pkg keywords and save the values located next to these keywords (resp., 0.2 Ω, 7 nH, and 1,5 pF).

When the scanning phase is completed, the program generates the SystemC-AMS file. This file includes one top level module with the correct number of I/Os. Each I/O is instantiated as a separate module inside the top level component and can be directly associated with a SystemC digital simulation model (see Figure 1): on one hand, a signal on an output pin is produced with the help of a switch component, commanded by the digital model, on the other hand, a voltage on an input pin is converted by a threshold detector to generate a boolean value that is sent to the digital model. Obviously, each I/O module models the electrical circuits presented above, with the extracted values of the R, L, C components. Algorithm 3 presents a sample of an in-out pin module created from an IBIS file specification.

This tool was used to generate a SystemC-AMS model out of the IBIS description of the CAT24WC0 2 Kbit I^{2}C EEPROM chip from catalyst semiconductors [22]. This model serves as the analog interface of both slave memory devices featured in the I^{2}C platform.

3.3. Transmission line model

The bus lines feature a number of imperfections and can suffer from any kind of signal integrity perturbation. This section presents how the electrical model of the bus lines can be included in our modelling platform.

So far, we have considered the bus lines as perfect conductors immune to any kind of interference. Still, the nature of the wires as well as their geometry can modify the behaviour of the transmitted signals and, of course, so does the external environment. These imperfections are represented with R, L, C elements that we integrate in our platform. Our line model below (Figure 14) is based on [23].

Each bus line includes the wire resistance and also an inductance that represents the wire self-inductance. The value of this inductance is given by the following formula, where l is the wire length and d the wire diameter in centimetres:

$$L = 0.002l\left(\ln\left(\frac{4l}{d}\right) - 0.75\right)\ [\mu\text{H}]. \quad (1)$$

In the situation of two (or more) adjacent lines, which is likely to appear in the case of the I^{2}C bus, a mutual coupling phenomenon appears. It is represented by a parallel

```
struct IBIS_INOUT_PIN : sc_module
{
   sc_out<bool>   i; // Input signal sent to digital block
   sc_in<bool>    o; // Output signal sent by digital block
...
   sca_elec_port  pin; // Inout Pin
...
   sca_sdf_signal<double> in_sdf; // Pin Voltage
...
   sca_elec_ref   gnd;    // Electrical Ground
   sca_elec_node B;       // Node between R_pkg & L_pkg
   sca_elec_node A;       // Node between R_pkg & L_fix
   sca_elec_node V;       // Node between R_fix & switch
...
   // Declaration of IBIS RLC elements
   sca_r *R_pkg, *R_fixture; sca_c *C_pkg, *C_fixture;
   sca_l *L_pkg; sca_vconst *V_fixture;
   sca_rswitch *sw1;   sca_vd2sdf *conv_pin;
...
   // Threshold converts input voltage to bool value
   THRESHOLD_DETECTOR *thd_in;
...
   SC_HAS_PROCESS(IBIS_INOUT_PIN);
...
   // Constructor: RLC values retrieved from IBIS file
   IBIS_INOUT_PIN( sc_module_name insname,
         double R_pkg_value, double C_pkg_value,
         double L_pkg_value, double C_fixture_value,
         double R_fixture_value, double V_fixture_value )
      {
         // PACKAGE ELEMENTS INSTANCIATION
            C_pkg->value = C_pkg_value;
            C_pkg->p(pin); C_pkg->n(gnd);
...
            L_pkg->value = L_pkg_value;
            L_pkg->p(B);   L_pkg->n(pin);
...
            R_pkg->value = R_pkg_value;
            R_pkg->p(A);   R_pkg->n(B);
...
         // PACKAGE ELEMENTS INSTANCIATION
            C_fixture->value = C_fixture_value;
            C_fixture->p(A); C_fixture->n(gnd);
...
            R_fixture->value = R_fixture_value;
            R_fixture->p(Vfix); R_fixture->n(B);
...
            V_fixture->value = V_fixture_value;
            V_fixture->p(Vfix); V_fixture->n(gnd);
...
         // Switch controlled by digital block
            sw1->p(V); sw1->n(gnd);
            sw1->ctrl(o); sw1->off_val = true;
...
         // Threshold: Pin Voltage is sent to digital
         // block as bool value
            thd_in->threshold = 1.1;
            thd_in->in (in_sdf); thd_in->out(i);
            };
         }
```

Algorithm 3: SystemC-AMS code sample generated from IBIS specification.

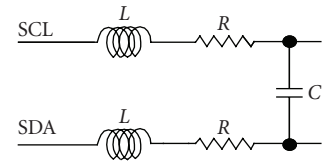


Figure 14: Transmission line model.

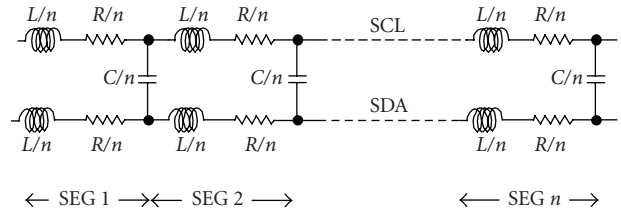


Figure 15: Transmission line model in n segments.

capacitance whose value depends on the diameter d of the wires and the distance D between them. The mutual capacitance is given by the following formula:

$$C = \frac{\pi 0.0885}{\cosh^{-1}(D/d)}\ [\text{pF/cm}]. \tag{2}$$

In a first step, we considered that the mutual inductance between the I^2C wires is negligible.

The length of the bus lines, for instance in an automotive context, can be quite important, up to several metres. Therefore, the global line model of Figure 14 can be divided into several segments, such as in Figure 15. With this representation, we can take into account a distance variation between two bus lines, affect only a part of the bus with a perturbation, or simply use segments with different lengths to respect the node topology in the system (Figure 16).

Our SystemC-AMS line model comes in the form of one top module that instanciates n segment submodules. The length of each segment can be parameterized in the submodule constructor (Algorithm 4).

4. PLATFORM SIMULATION

We present in this section the simulation of our field-bus-based platform, summarized in Figure 17. The SystemC and SystemC-AMS description allow us to cosimulate the MIPS or the 8051 embedded software with the hardware behaviour of the components, whether analog or digital. This section first gives some information about the code that is executed during simulation, then shows its impact on the digital blocks of the platform, especially the bus controllers. We focus next on the analog response on the bus, presenting the influence of the transmission line on the shape of the signal. Finally, we provide simulation performances for our mixed platform.

4.1. Embedded code

The aim of our simulation is to validate the functionality of the bus controller by testing all the protocol features.

```
struct LINE_I2C:sc_module
{
   sca_elec_port scl1, scl2;      // Segment ends for SCL
   sca_elec_port sda1, sda2;      // Segment ends for SDA
   sca_elec_node lr1, lr2;        // Node between L and R
...
   double L_value, R_value, C_value; // Elements value
...
   sca_r *R_scl, *R_sda;
   sca_l *L_scl, *L_sda;
   sca_c *C_cpg;
...
   SC_HAS_PROCESS(LINE_I2C);
...
   LINE_I2C(sc_module_name insname, double length)
      {   // Length in cm
          L_value = 7.8e − 9*length;
          R_value = 0.5*length;
          C_value = 0.1e − 12*length;
...
      // Elements Instanciation
...
          L_scl->p(scl1); L_scl->n(lr1);
          L_scl->value = L_value;
...
          R_scl->p(lr1); R_scl->n(scl2);
          R_scl->value = R_value;
...
          L_sda->p(sda1); L_sda->n(lr2);
          L_sda->value = L_value;
...
          R_sda->p(lr2); R_sda->n(sda2);
          R_sda->value = R_value;
...
          C_cpg->p(scl2); C_cpg->n(sda2);
          C_cpg->value = C_value;
      };
   };
```

Algorithm 4: Line segment SystemC-AMS code sample.

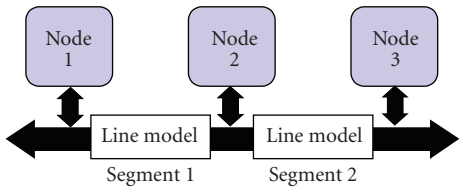


Figure 16: Line model and system topology.

Therefore, the embedded code found on the two master devices mainly consists in bus access operations.

The following assembler code (Algorithm 5) is compiled and stored in the internal ROM of the 8051 microcontroller. When it runs, the program performs an I²C write operation: the data byte 4Eh is written on the B4h cell address of the slave RAM device.

This write operation takes place in two stages: first the IP sends the cell address to the slave device (the A0 h value is made of the 50 h 7-bit slave address and the R/W bit set to 0), then it writes the data byte to this same slave device.

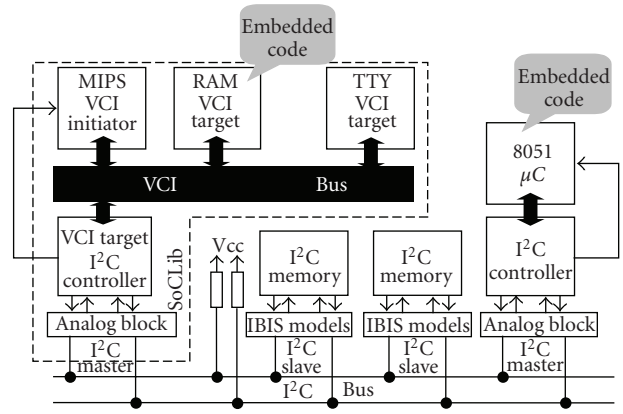


Figure 17: I²C platform.

```
        //      Write in RAM(slave address = 0x50)
        //      the value 0x4E in cell 0xF1
...
main: MOV     R0, #A0 // Put slave address in R0 register
      MOV     A, #B4 // Put cell address in A register
      MOVX    @RO, A // Send R0 + A to I2C IP
      MOV     R0, #A0 // Put slave address in R0 register
      MOV     A, #4E // Put data byte in A register
      MOVX    @RO, A // Send R0 + A to I2C IP
```

Algorithm 5: 8051 assembler code sample—byte writing operation.

Similarly, Algorithm 6 gives a sample of the MIPS code, featuring the assembler functions used to request the controller IP a read or a write on the I²C bus, as well as the interrupt subroutine which is called when a read data has been received from the I²C bus and is available for the MIPS master.

4.2. Digital simulation

When running, the codes found in the SoC and the microcontroller send requests and therefore activate their respective I²C controller. In particular, we can see in Figure 18 a sequence featuring a write operation requested by the 8051 to one of the RAM devices followed by an MIPS-commanded read request of the same RAM cell address.

The SystemC chronogram shows the 8051 positioning the 4 Eh byte on its data bus and the F1 h value on its address bus, resulting on a first transmission on the I²C bus, then the SoC bus controller performing a second transmission to retrieve the data from the RAM component. The chronogram shows the digital behaviour of the SDA bus line (featured on the last row), as well as the SDA commands sent by each component, indicating which device has control of the bus at any given time. We can also notice that an IRQ

```
// Interrupt Subroutine
void SwitchOnIt(int it)
{
   int i;
   :
// Identify the active interrupt of highest priority
   for (i = 0; i < 8; i++) if (it&(1 << i)) break;
   :
   switch (i)
   {
      case 0: break;       // SwIt 0
      case 1: break;       // SwIt 1
      case 2: read_interface(); break; // It 0
      case 3: break;       // It 1
      case 4: break;       // It 2
      case 5: break;       // It 3
      case 6: break;       // It 4
      case 7: break;       // It 5
      default: break;
   }
}
   :
// Main Program
int main(void)
{
   write_byte(0x50, 0x57, 0x69);
   // Parameters = Slave Adr (stored in $4 register)
   //              Cell Adr (stored in $5 register)
   //              Data Byte (stored in $6 register)
   :
   read_byte(0x51, 0xf1);
   // Parameters = Slave Adr (stored in $4 register)
   //              Cell Adr (stored in $5 register)
   :
   while (1);
   return 0;
}
   :
// Byte Writing ASM Code Sample
write_byte:
   la   $3,0xc0000000   // Load VCI target (I2C IP)
                        // address to $3 register
   sll  $4, $4, 9       // Slave adr in high part of register
   or   $4, $4, $5      // Cell adr in low part of register
   sh   $4, 0 ($3)      // (save half-word)
                        // send target + cell adr to IP
   sb   $6, 1 ($3)      // Send data byte to IP
   nop
   j    $31             // return from subroutine
.end write_byte
// Byte Reading ASM Code Sample
read_byte:
   :
   la   $3,0xc0000000   // Load VCI target (I2C IP)
                        // address to $3 register
   sll  $4, $4, 9       // Slave adr in high part of register
   ori  $4, $4, 0x100   // Set R/W Bit to Read
   or   $4, $4, $5      // Cell adr in low part of register
   sh   $4, 0 ($3)      // save half-word ->
   nop                  // send target + cell adr to IP
   j    $31             // return from subroutine
```

Algorithm 6: MIPS code sample.

signal is generated when the 4Eh value is available in the MIPS I^{2}C controller.

4.3. Analog simulation

The chronogram in Figure 19 represents the analog behaviour of the I^{2}C bus lines during a write transmission issued by the MIPS-based SoC. It does not take into account the transmission line model. On the first row of the chronogram, the slave address (50 h) is first sent on the bus, then on the second row, after an acknowledge from this slave device, the master transmits the cell address (B4 h) it wants to write into. Finally, on the third row, after another acknowledge from the slave, the data byte (4 Eh) is sent to the slave, followed by a last acknowledge from the slave and a stop bit, signaling the end of transmission.

The chronogram also displays the shape of the analog signal. The rising and falling times that can be seen depend on the analog interface of the component that has control of the bus when a particular bit is transmitted. In the case of the chronogram in Figure 19, most of the bits are issued by the 8051 bus controller interface, except for the acknowledges which are issued by the IBIS-converted interface of the RAM device.

Our platform also allows us to test the multimaster arbitration of the I^{2}C bus as it is implemented in our model. In the chronogram shown in Figure 20, both masters send a start bit at the same moment. The SDA bus line follows the two commands as long as they are identical, and when they differ, the master which sends a 0 bit (in this case the MIPS) takes control of the bus and the 8051 stops transmitting on SCL and SDA. Also of interest is the wired-AND function performed on the SDA and SCL lines of the bus. This shows that our mixed-signal controller model successfully performs electrical multimaster arbitration.

Finally, Figure 21 presents the signal behaviour if we take into account the line imperfections with the model presented in Section 4. We can clearly see the effect of the wire inductance and the mutual capacitance.

4.4. Simulation performances

Table 1 presents simulation speeds for various configurations of the simulation platform presented in Section 4. The measurements were performed on a Linux workstation equipped with a Pentium M processor running at 1.73 GHz, a L2 cache of 2 Mb, and 1 Gb of SDRAM. Sampling period for the analog models is 100 nanoseconds.

Our aim is to measure the increase of simulation time between a purely digital simulation in SystemC of a platform, and a mixed SystemC/SystemC-AMS simulation of the same platform. We first measure this overhead in the case of a simplified platform made of a microcontroller node and a slave RAM, first without and then with the transmission line model. Results show that for the 8051 platform, the overhead due to the AMS parts of the model is relatively important, though simulation is still very fast.

In the second case that is the full 4-node platform including the transmission line model, the impact is less

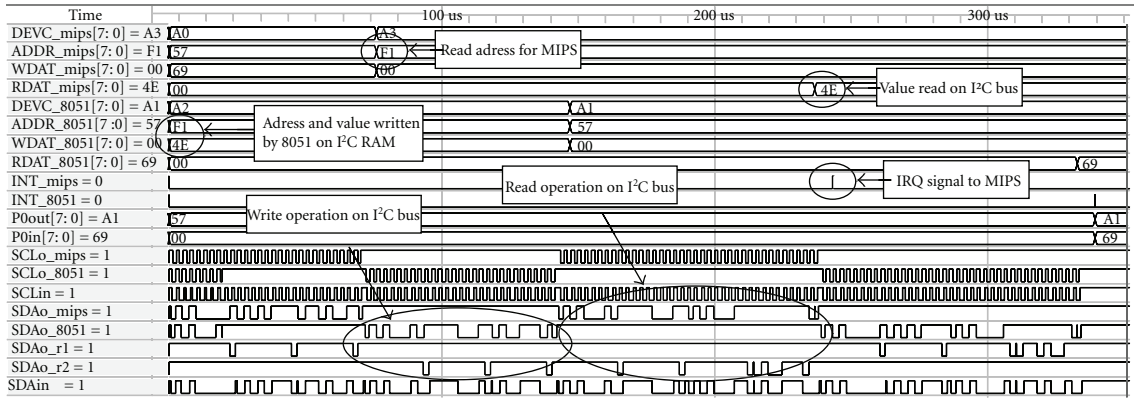


Figure 18: Simulation with 8051 and MIPS master.

Table 1: I²C Platform simulation performance in cycles/s.

Application	SystemC platform	Sys.C + Sys.C-AMS equivalent plarform	Overhead
8051 node + RAM with IBIS	149 300	106 400	29%
8051 node + RAM with IBIS + line model	149 300	87 500	41%
4-node platform without line model	25 800	23 500	9%
Complete I²C platform (Figure 19)	25 800	19 500	24%

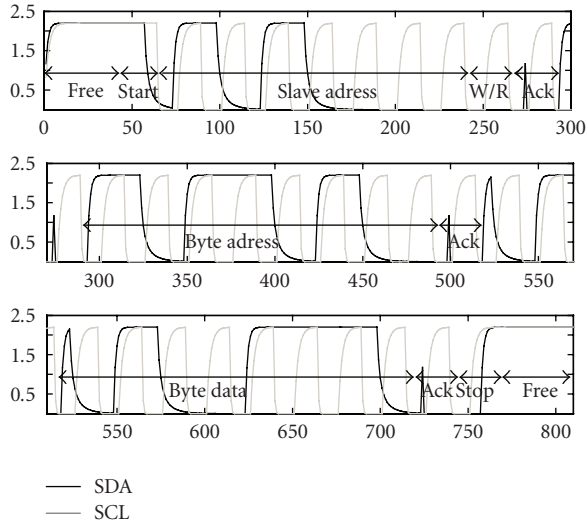


Figure 19: Analog simulation—byte writing operation.

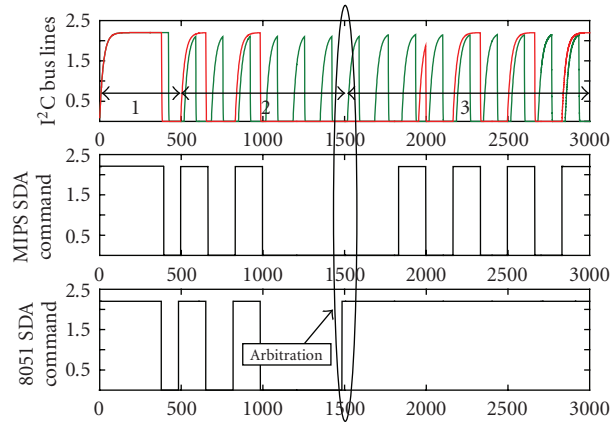


Figure 20: Multimaster mode arbitration.

prominent, mainly because the proportion of purely SystemC components is more important. In this latter configuration, analog simulation of the I²C bus is performed only 9% slower than a digital-only simulation if we do not include the line model, and 24% if we add a line segment model between two nodes.

5. CONCLUSION

In this paper, we showed how to model field-bus communications in the context of embedded systems. We described a heterogeneous model that features three main parts. Firstly, an I/O bus controller interfaces the processor cores with the field bus. Secondly, an analog part represents the interface of the I/O controller with the bus lines. To accurately model the analog behaviour of the components, we introduced an IBIS-to-SystemC-AMS converter to derive the analog description of the field bus nodes analog interface from industrial circuits. Thirdly, we modelled the bus lines as an array of segments that take into account the electrical imperfections of the bus lines.

Simulations showed the successful operations of the I²C field bus. The SystemC-AMS simulation times showed different overheads over digital SystemC: 40% for a 8051 to

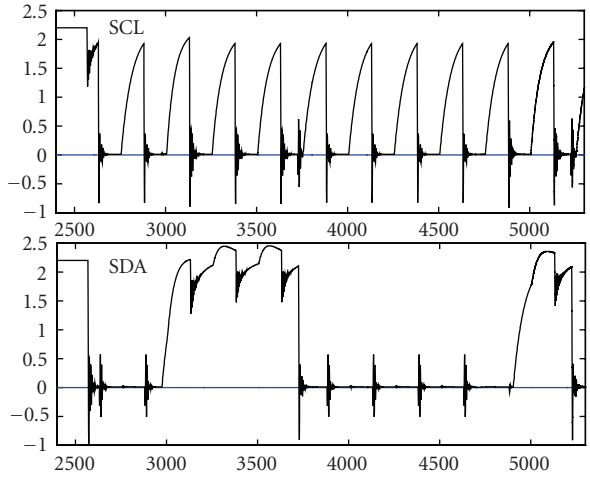


Figure 21: Bus lines simulation with transmission line model.

RAM I^{2}C communication, and around 25% for an MIPS-based SoC plus 8051 system. This overhead is acceptable for a mixed-signal simulation.

The benefits of our approach are that it includes analog and digital models in a single environment. It will therefore allow to simulate interactions between analog and digital functions that are usually discovered on the hardware prototypes.

As a next step, we intend to show that our approach is generic by applying it to other field busses such as CAN. Further, this SystemC-AMS simulation platform stresses the interest of this language as a unique tool from the specification to the verification of AMS embedded systems, gathering together software execution with analog and digital behaviour.

REFERENCES

[1] M. Chiodo, D. Engels, P. Giusto, et al., "A case study in computer-aided co-design of embedded controllers," *Design Automation for Embedded Systems*, vol. 1, no. 1-2, pp. 51–67, 1996.

[2] J. Liu, X. Liu, and E. A. Lee, "Modeling distributed hybrid systems in Ptolemy II," in *Proceedings of the American Control Conference (ACC '01)*, pp. 4984–4985, Arlington, Va, USA, June 2001.

[3] A. Vachoux, C. Grimm, R. Kakerow, and C. Meise, "Embedded mixed-signal systems: new challenges for modeling and simulation," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '06)*, pp. 991–994, Island of Kos, Greece, May 2006.

[4] J. Oudinot, J. Ravatin, and S. Scotti, "Full transceiver circuit simulation using VHDL-AMS," in *Proceedings of Forum on Specification and Design Languages (FDL '02)*, pp. 29–33, Marseille, France, September 2002.

[5] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[6] A. Vachoux, C. Grimm, and K. Einwich, "SystemC-AMS requirements, design objectives and rationale," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, pp. 388–393, Munich, Germany, March 2003.

[7] A. Vachoux, C. Grimm, and K. Einwich, "Extending SystemC to support mixed discrete-continuous system modeling and simulation.," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '05)*, vol. 5, pp. 5166–5169, Kobe, Japan, May 2005.

[8] A. Vachoux, C. Grimm, and K. Einwich, "Towards analog and mixed-signal SOC design with systemC-AMS," in *Proceedings of the 2nd International Workshop on Electronic Design, Test & Applications (DELTA '04)*, pp. 97–102, Perth, Australia, January 2004.

[9] A. Vachoux, C. Grimm, and K. Einwich, "Analog and mixed signal modelling with SystemC-AMS," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '03)*, vol. 3, pp. 914–917, Bangkok, Thailand, May 2003.

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

[11] OSCI Analog/Mixed-signal Working Group (AMSWG), http://www.systemc.org/apps/group_public/workgroup.php?wg_abbrev=amswg.

[12] O. Romain, T. Cuénin, and P. Garda, "Design and verification of mixed-signal I/O IPs: an 12C bus controller," in *Proceedings of IEEE International Symposium on Industrial Electronics (ISIE '04)*, vol. 1, pp. 77–81, IEEE Press, Ajaccio, France, May 2004.

[13] M. Alassir, J. Denoulet, O. Romain, and P. Garda, "A SystemC AMS model of an 12C bus controller," in *Proceedings of IEEE International Conference on Design and Test of Integrated Systems in Nanoscale Technology (DTIS '06)*, pp. 154–158, IEEE Press, Tunis, Tunisia, September 2006.

[14] M. Alassir, J. Denoulet, O. Romain, and P. Garda, "Modeling I^{2}C communication between SoCs with SystemC-AMS," in *Proceedings of IEEE International Symposium on Industrial Electronics (ISIE '07)*, pp. 1412–1417, IEEE Press, Vigo, Spain, June 2007.

[15] M. Alassir, J. Denoulet, G. Vasilescu, O. Romain, R. Arnaud, and P. Garda, "Modeling field bus communications for automotive applications," in *Proceedings of the Forum on specification and Design Languages (FDL '07)*, Barcelona, Spain, September 2007.

[16] Philips Semiconductors, The I^{2}C-Bus Protocol Specification, Document Order Number: 9398 393 40011, January 2000, http://www.nxp.com/acrobat/literature/9398/39340011_21.pdf.

[17] Philips Semiconductors, PCF 8584, I2C bus controller, http://www.nxp.com/acrobat/datasheets/PCF8584_4.pdf.

[18] SoCLIB, "A modelisation & simulation plat-form for system on chip," 2003, http://www.soclib.fr/.

[19] VSI Alliance, Virtual Component Interface Standard—version 2 (OCB 2 2.0), http://www.vsi.org/index.htm.

[20] Intel, 80C51 Single-Chip 8-bit Microcontroller datasheet, ftp://download.intel.com/MCS51/datashts/27050108.pdf.

[21] IBIS (I/O Buffer Information Specification), Version 4.2, June 2006, http://vhdl.org/pub/ibis/ver4.2/ver4_2.pdf.

[22] CAT24WC01/02/04/08/16, 1K/2K/4K/8K/16K-Bit Serial EEPROM datasheet, Catalyst Semiconductors, http://www.catsemi.com/datasheets/24WC.

[23] G. Vasilescu, *Electronic Noise and Interfering Signals: Principles and Applications*, chapter 12, Springer, New York, NY, USA, 2005.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 171358, 9 pages
doi:10.1155/2008/171358

Research Article

Novel Methodology for Functional Modeling and Simulation of Wireless Embedded Systems

Emma Sosa Morales, Giorgia Zucchelli, Martin Barnasconi, and Nitasha Jugessur

NXP Semiconductors, Corporate Innovation and Technology, High Tech Campus 37, 5656AE Eindhoven, The Netherlands

Correspondence should be addressed to Emma Sosa Morales, emma.sosa.morales@nxp.com

Received 12 October 2007; Accepted 8 April 2008

Recommended by Christoph Grimm

A novel methodology is presented for the modeling and the simulation of wireless embedded systems. Tight interaction between the analog and the digital functionality makes the design and verification of such systems a real challenge. The applied methodology brings together the functional models of the baseband algorithms written in C language with the circuit descriptions at behavioral level in Verilog or Verilog-AMS for the system simulations in a single kernel environment. The physical layer of an ultrawideband system has been successfully modeled and simulated. The results confirm that this methodology provides a standardized framework in order to efficiently and accurately simulate complex mixed signal applications for embedded systems.

1. INTRODUCTION

Today's telecommunication systems embed high performance analog, mixed-signal, radio frequency (RF), and digital circuitry in a single chip. An example of such an embedded system is the NXP semiconductors wireless USB end-to-end silicon solution based on the ultrawideband (UWB) technology [1].

UWB systems operate at radio frequency in the GHz-range and use advanced modulation methods such as orthogonal frequency-division multiplexing (OFDM), fast frequency hopping techniques, smart radio control, and calibration to maximize link reliability and channel efficiency but also to minimize interference with other services. Growing complexity in the RF front end as well as very accurate digital signal processing is observed. However, the analog and digital subsystems are often developed separately. The combination and tight interaction of RF implementation with the baseband algorithms, required by calibration and control loops, make the design and simulation of such systems a real challenge.

The goal of this paper is to present a methodology for the modeling and the simulation of the physical layer (PHY) of a wireless system within a single simulation environment. This framework enables the validation of the complete physical layer, taking into account the interaction between the analog and the RF models with the baseband algorithms described in the C language.

This paper is organized in sections. In Section 2, the challenges of next generation wireless systems are presented, introducing the UWB as a relevant example. In Section 3, the work related to the applied methodology is summarized. In Section 4, the methodology is detailed. In Section 5, an overview of the behavioral models for the UWB PHY is given. In Section 6, the use of the Verilog Procedural Interface (VPI) as part of the methodology is explained. In Section 7, the results are presented for the UWB PHY followed by the conclusions.

2. PROBLEM DESCRIPTION

In this section, the challenges of today's wireless systems are described and divided into two categories: design challenges, related to the technology and implementation of such systems, and the subsequent modeling and simulation challenges. The requirements that have to be fulfilled by a design and verification methodology are mainly dictated by the simulation challenges of these wireless embedded systems.

2.1. Design challenges

Next generation wireless systems are difficult to design and verify due to extremely demanding requirements such as high transmission frequencies, large bandwidth, sophisticated modulation and coding techniques for high data rate, fast frequency hopping, and low or strictly controlled transmission power. All these features are present in the UWB communication system.

WiMedia proposes the technical specifications for the media access control and the physical layer of the UWB system [2]. A multiband orthogonal frequency division modulation (MB-OFDM) scheme is specified for the physical layer. The information is transmitted in packets which are composed of a variable number of OFDM symbols according to the achievable data rate. A total of 110 subcarriers is used to transmit one OFDM symbol with a channel bandwidth of 528 MHz.

The available spectrum is divided into 14 bands which cover the frequency range from 3.1 GHz to 10.6 GHz. To allow transmissions covering such a large bandwidth, regulation bodies for different geographical regions have put in place severe broadcast restrictions for power spectral emissions of spurious and other interferences. By doing so, UWB devices can make use of an extremely wide frequency band while not emitting enough energy to be noticed by narrowband devices nearby. Due to the restrictions regarding power spectral emissions, the transmitted spread spectrum signal has similar characteristics as wideband noise. Therefore, sophisticated signal processing algorithms and receivers with high sensitivity are required to recover the information especially in the presence of interferers. Furthermore, to increase the robustness, the UWB communication system adopts fast frequency hopping across different frequency bands.

2.2. Modeling and simulation challenges

The design challenges become true modeling and simulation challenges when the physical layer has to be validated. The high frequencies impose very small time steps for the transient simulation which lead to long simulation times. The approach of equivalent baseband modeling [3] does not help to reduce the simulation time due to the large bandwidth of the signals. The complex signal processing and large channel bandwidth increase to a large extent the amount of data that has to be handled by the simulator. In-band and out-of-band interferences and the contribution of noise also add to the complexity of the simulation.

The conventional approach for the design of complex systems consists in dividing it into subsystems, each with limited complexity. The subsystem specifications are defined during the system exploration phase and are then passed to different implementation teams. The design of these subsystems is done by teams of specialists, often distributed all over the world, using multiple and dedicated tools with the consequence that the overall performance of the system is not always monitored. Overspecification often results from adopting a conservative approach to design the different subsystems. Correction loops across multiple domains make the system partitioning critical, requiring a strict definition of the interfaces. Moreover, it is extremely difficult to prove before the tape-out that the designed subsystems fulfill the given requirements in realistic conditions (e.g., including interferers and noise).

Another difficulty encountered when modeling and simulating the whole system is due to the use of multiple languages and simulation environments. Apart from providing a single simulation environment, a system design and verification methodology should also fulfill the following requirements.

(i) Single kernel solution: synchronization problems due to the interaction between different simulation kernels are avoided, resulting in better performance and easier debugging.

(ii) Library independent: models described in standardized languages can be used, thereby reducing the dependency on vendors' libraries. The user has the freedom to decide the level of detail implemented in the model. Refinements to include nonideal effects during both top-down design and bottom-up verification are possible.

(iii) Tool independent: the implementation of the methodology should be compatible with existing mixed signal and mixed abstraction level design environments and flows offered by commercial tool vendors.

(iv) Reasonably fast: tradeoffs between modeled impairments, accuracy, and simulation speed should be possible in the simulation framework for both RF and baseband models.

(v) Aligned with the design community: existing design methodologies, modeling approaches, and standardized languages used by the system and circuit design community should be supported.

(vi) Support vector processing: to handle OFDM and other complex modulation techniques, vector processing should be facilitated by the simulation framework.

The next section gives an overview of the different solutions available to address the simulation challenges described.

3. RELATED WORK

The complexity of today's telecommunication systems triggers electronic design automation tool vendors and universities to provide solutions or study alternatives, ranging from supporting multiple description languages, combining dedicated solvers, or even linking different environments together.

In the past years, significant effort has been spent to develop mixed signal tools combining digital and analog solvers in a single framework and to support mixed signal languages [4, 5]. Nowadays, most major tool vendors provide a solution for the circuit implementation, modeling, and verification of mixed signal applications [6–8].

Most recently, the main effort to tackle the system complexity has been dedicated to combining different levels of design abstraction, facilitating both top-down design and bottom-up verification [9–11]. MATLAB and the C language are often used for the definition of the system specifications and are widely recognized as the languages for system level design [12]. The lack of connection between the dedicated tools used for system level design and the ones used for circuit level implementation has lead to the trend of cosimulation support. Most commercial system level tools have an interface to C language and MATLAB [13–15] and provide cosimulation solutions with mixed signal integrated circuit (IC) design simulators. The list of available links among different tools and environments can easily get lengthy including all possible flavors of analog, digital, and RF solvers. Some examples include MathWorks Simulink and Cadence AMS Designer or Mentor Graphics AdvanceMS, Agilent ADS-Ptolemy and Cadence AMS Designer or Agilent Circuit Envelope, Agilent ADS-Ptolemy and Cadence NcSim or Mentor Graphics ModelSim, CoWare SPD and Cadence AMS Designer.

However, the configuration of the environment and the test bench for the cosimulation is often not trivial and requires extra effort from the architect or the designer. Moreover, converter blocks are often necessary to enable the communication between the simulators [16]. These converter blocks make the interface less transparent. The tight communication between different solvers needed in a cosimulation approach is a potential source of synchronization problems, deteriorating the simulation performances in terms of speed and increasing the difficulty of the debugging process.

A single kernel solution linking system level with mixed signal IC design presents advantages over cosimulation solutions [3, 17]. The possibility of using standardized programming interfaces to call C code functions within hardware description languages (HDL) models overcomes the limitations of cosimulation and guarantees a transparent definition of the simulation set up. To the authors knowledge, the standardized interfaces [18, 19] have since long been used to set up cosimulation with third-party tools [10] and for testing purposes [20, 21]. The innovative approach described in the next section proposes the use of such interfaces to embed digital signal processing in a mixed signal environment without the penalties of cosimulation.

4. APPLIED METHODOLOGY

In this section, the methodology to address the modeling and simulation challenges is introduced.

The methodology is based on the following three elements:

(1) the use of a mixed signal simulator to combine analog and digital signals in a single kernel environment;

(2) the use of behavioral models to describe the functionality of the RF and mixed signal blocks with standardized languages and the use of high-level languages (C/C++) for algorithms;

(3) the use of an environment that allows functional description in combination with circuit level implementation to enable mixed abstraction level simulations.

The mixed signal simulator provides a framework where analog and digital seamlessly come together. Such a framework is obviously useful for the simulation of blocks described at implementation level as transistor netlists and RTL. Most behavioral languages are supported by mixed signal simulators, allowing both top-down and bottom-up design methodologies: systems can be modeled and simulated at different levels of design abstraction.

Through standardized languages, dedicated behavioral models can be developed extending the commercially available libraries. Since the simulation speed depends on the amount of details captured (e.g., impairments), models can be refined to achieve the desired compromise between accuracy and simulation speed. In-house developed models are intellectual property (IP), representing a valuable asset that can be exchanged among design groups, thereby increasing the level of expertise and IP reuse. Examples of behavioral models for the functional description of the UWB RF transceiver are presented in Section 5.

In a mixed signal IC design environment, Verilog and Verilog-AMS are often used to model the digital and analog implementation. However, for more complex systems such as OFDM modems, where many computations are done using vector or matrix operations, these HDL languages do not offer sufficient semantics for efficient functional modeling [3, 17]. Languages such as C or C++ are often used for this purpose. The Verilog Procedural Interface (VPI) [18] allows calling algorithm descriptions in C language from within Verilog or Verilog-AMS modules. In Section 6, the interaction between a given C function and a Verilog or Verilog-AMS module through the VPI is described.

Since the VPI is part of the IEEE 1364 standard for Verilog HDL and supported by most tools vendors, the applied methodology is tool-independent and can be implemented in several commercially available simulation environments.

Figure 1 shows the philosophy behind the methodology. Within a single environment, C language, behavioral models, transistor netlists, and RTL can be brought together. This methodology offers a framework in which the challenges of next generation communication systems can be handled. The standardized VPI for mixed abstraction level simulations combined with the standardized HDLs for behavioral modeling and a mixed signal environment results in a differentiating methodology compared to other solutions available on the market.

5. MODELING THE UWB PHYSICAL LAYER

In this section, the models for each block of the UWB PHY and their interfaces are described. The baseband algorithms, compliant with the WiMedia specification, are first explained. The functional description of each constituent of the RF transceiver is then summarized. These models have simple characteristics and introduce only fundamental

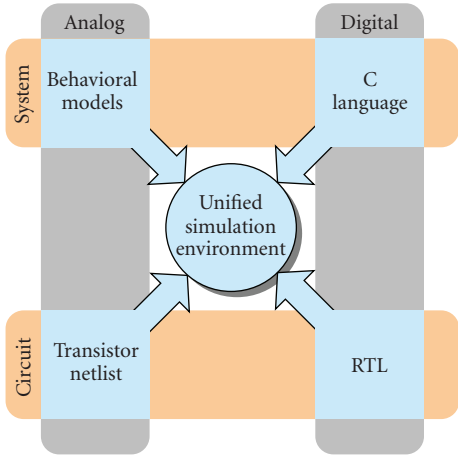


Figure 1: With the applied methodology, different levels of design abstraction for the analog and digital domains come together in the same framework.

impairments. Within the framework, further refinement of the models is possible.

Figure 2 shows the block diagram of the UWB PHY. Behavioral models have been developed in Verilog-A or Verilog-AMS for the RF transceiver and in C language for the baseband. These models are combined together in a single test bench for system validation. Within the same test bench, the functional models can be replaced by transistor level descriptions for circuit-level validation in the system context. This framework therefore facilitates complex mixed signal and mixed abstraction level simulations.

5.1. Baseband models

The *baseband transmitter* generates the OFDM signal compliant to the physical layer specification of the WiMedia proposal. This requires algorithms for digital signal processing: scrambling, forward error correction, code interleaving, constellation mapping, OFDM modulation using inverse fast Fourier transformations, insertion of pilots, guard carriers addition, and cyclic prefix addition.

This functional model is described as a C code function that generates two vectors representing one frame of the in-phase I and quadrature Q components of the OFDM signal. It also generates the signal which controls the local oscillator frequency of the *IQ Modulator*. The transmission rate and the length of the packet are among the parameters of the baseband transmitter model.

The *baseband receiver* implements the inverse operations of the baseband transmitter. In order to recover the transmitted data, the following procedures are used: timing and frequency offset correction, OFDM demodulation using fast Fourier transformations, channel correction, constellation demapping (digital demodulation), deinterleaving, forward error correction decoding and descrambling.

The functional model of the baseband receiver is implemented as a C code function. It takes as inputs the I and Q components of the OFDM signal and the information passed from the synchronizer in order to extract the payload from the packet. The model has the same parameters as the baseband transmitter.

The *synchronizer* calculates, per sample of the I and Q components, the information needed by the baseband receiver to identify the payload in the packet. The synchronizer also recovers the hopping sequence and controls the local oscillator frequency of the *IQ demodulator*.

The functional model of the synchronizer is implemented in C language and does not have any user-defined parameters.

5.2. RF transceiver models

The *IQ modulator* and *demodulator*, respectively, up- and down-convert the I and Q samples with a carrier frequency that changes from symbol to symbol. A defined hopping sequence is used to generate the carriers that drive the mixer for the frequency conversion.

The Verilog-AMS models of the modulators [22] include nonidealities such as third-order distortion and gain compression. To simplify the model interface, the local oscillator is described as an internal signal. The parameters of the model are the input impedance, output impedance, voltage conversion gain, and input referred third-order intercept point. The demodulator model also contains a low pass filter to reject the signal image of the demodulated signal at the higher frequency.

The *TX filter* and the *RX filter*, respectively, filter the symbols generated by the baseband transmitter and recovered by the demodulator.

The Verilog-AMS models of the filters implement a linear Laplace function with poles and zeros as arguments. The poles and zeros are extracted from the circuit implementation after a pole-zero analysis. Input and output impedances are specified as parameters.

The *PA* and *LNA* amplify the signal power to compensate for the channel attenuation.

The Verilog-A model of the amplifiers includes second-order distortion, third-order distortion, and gain compression. The parameters of the models are the input impedance, output impedance, voltage conversion gain, input referred third-order intercept point, and input referred second-order intercept point.

The *channel* represents the pathway over which data is transferred from the antenna of the transmitter to the one of the receiver.

A Verilog-A model describes an ideal channel with the attenuation as a parameter. Interference, fading, multipath delay spread, and other nonidealities can be included in the model.

5.3. Interfaces

The signal types at the interfaces of the Verilog-AMS models are either *wreal* or *electrical*. Signals of type *wreal* are discrete in the time domain and are handled by the discrete event solver. *Wreal* is therefore preferred at the baseband interface. However, the *electrical* type is more appropriate for the

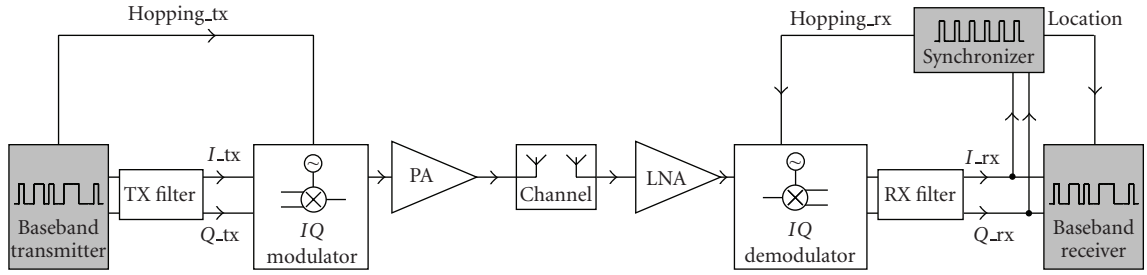


Figure 2: Block diagram of the UWB PHY with the baseband (in light grey) and the RF transceiver.

```
void vpi_wrapper ()
{
vpiHandle      taskHandle, argIter;
vpiHandle      inHandle, outHandle;
s_vpi_value    inVal, outVal;
double         *buffer;

taskHandle = vpi_handle(vpiSysTfCall, NULL);
argIter = vpi_iterate(vpiArgument, taskHandle);
if (argIter) {
   inHandle = vpi_scan(argIter);
   outHandle = vpi_scan(argIter);
   if (inHandle && outHandle ) {
      inVal.format = vpiIntVal;
vpi_get_value(inHandle, &inVal);
      /* Call to user-defined C function */
      user_func(inVal.value.integer, buffer);
      outVal.format = vpiRealVal;
      outVal.value.real = *buffer;
      vpi_put_value(outHandle, &outVal, NULL,
                vpiNoDelay);
      ...
}
```

Example 1

description of analog and RF signals. This type is handled by the analog solver (continuous time) which has to solve the differential equations from Kirchoff's laws at all electrical nodes.

6. EMBEDDING C CODE

This section describes the embedding of C code in a mixed signal IC design environment. The interaction between a C code function and a Verilog or Verilog-AMS model is through the Verilog procedural interface. The main features of the VPI are described in Section 6.1. The introduction of the notion of time within the VPI and the handling of the event synchronization and the vector processing for the UWB PHY are detailed in Section 6.2.

6.1. Verilog Procedural Interface

The Verilog Procedural Interface is the programming interface for the Verilog hardware description language and standardized under IEEE 1364. The VPI allows the writing of applications to create simulator tasks or functions that can be called from within Verilog HDL designs. VPI applications are dynamically loaded by the simulator, thereby making the system task or function part of the simulator executable. These user-defined system tasks are called from the HDL design during simulation.

In the presented methodology, the VPI is used to embed C code functions describing the baseband algorithms in an IC design environment. All the functionalities and the flexibility of the C language with respect to dynamic vector allocation (pointers) are supported within the VPI making it suitable for the support of vector-based digital signal processing.

Figure 3 shows the interaction between a C code function and a Verilog or Verilog-AMS module through the VPI. The functionality to be imported is described as a C code function (inner kernel in Figure 3). By using the access and utility routines provided by the VPI, a wrapper is built around this C code function. The wrapper handles the inputs/outputs and allows the C code function to interact with the ports and variables of the Verilog-AMS module. Two functions are used for this purpose: *vpi_get_value()* and *vpi_put_value()*. *vpi_get_value()* is used to read a value from the Verilog-AMS module and to make it available to the C code function. *vpi_put_value()* allows the writing of the results calculated by the C code function to the Verilog-AMS module. Within the wrapper, any necessary signal type conversion is handled. No additional models is necessary to translate the signals as compared to cosimulation solutions where dedicated interfaces (e.g., connect modules) are often required.

In Example 1, the code in C language shows an example of how to combine the utility functions provided by the VPI in order to build the wrapper.

Once the wrapper is built, it needs to be registered as a simulator task or function and compiled into a shared object library. In our case, *gcc* has been used for compilation and linking and provides all standards options for debugging the C code function. The debugging possibilities of the VPI functions within the IC design environment are, however,

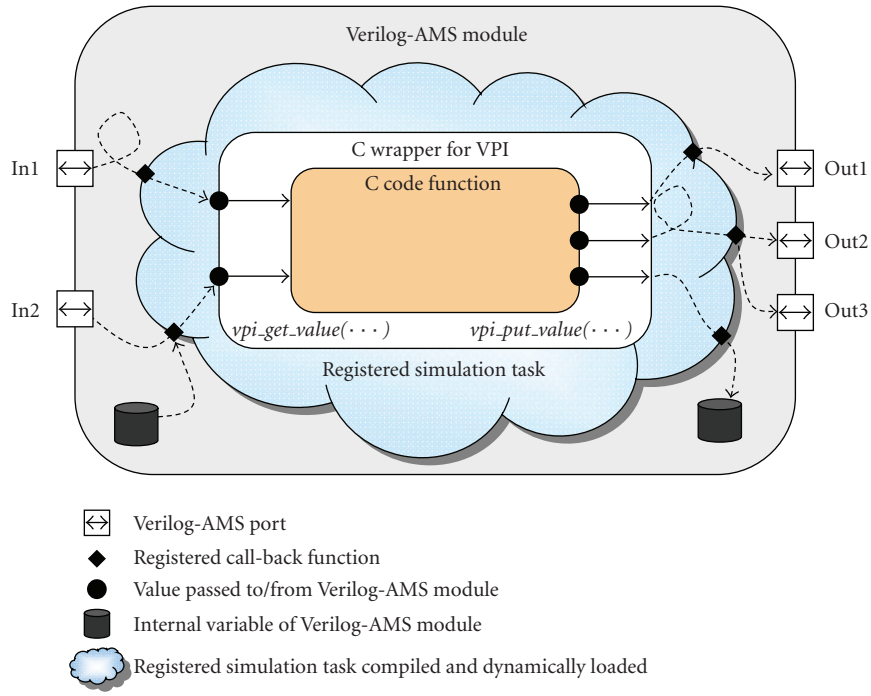


Figure 3: Schematic representation of the interaction between the Verilog-AMS module and the C code function wrapped using VPI routines.

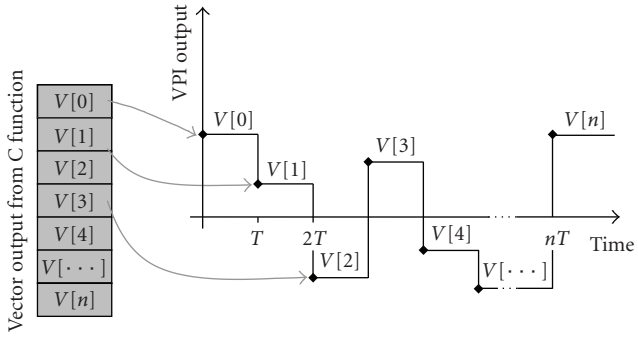


Figure 4: Process of unrolling the vector generated by the C code function in the time domain.

tool-dependent. The simulator task or function can be called from a Verilog or a Verilog-AMS module. The shared object is dynamically loaded during the elaboration phase.

In order to embed C code in a mixed signal IC design environment, the user should have basic knowledge of the C language as well as hardware description languages such as Verilog or Verilog-AMS. Familiarity with C development environments as well as with mixed signal simulators could also prove to be useful. The steps of creating the wrapper, registering the simulator task, compiling, and linking into a shared object could be fully automated by tool vendors in order to embed a C code function in a mixed signal IC design environment.

Table 1: Error on sampling time and characteristic frequency as a function of precision.

Precision	T (ns)	ΔT (ps)	Δf (MHz)
1 ns	2	106.060606⋯	29.568
1 ps	1.894	0.060606⋯	0.017

6.2. Time management

With the described methodology, the wrapper is used for introducing the concept of time that is otherwise not known in the C code function. For a simulation in the time domain, each sample is associated with a time stamp and is handled as an event by the digital solver. The events are detected with a finite accuracy. According to the chosen precision, a finite number of digits are used to represent the time. The timing error introduced with this approximated representation can be interpreted as a fixed jitter on the sampling time. In long simulation runs, this error can become relevant due to its cumulative behavior. Synchronization problems arise when events on a signal are missed due to the signal being generated and sampled with clocks having the same sampling rate but specified with different precisions.

The simulation time and the results depend on the chosen precision. For the UWB system, the sampling time of the baseband signals is 1.8939393⋯ nanoseconds (1/528 MHz), rounded to T by the digital solver. The resulting error ΔT in the time domain affects the signal representation in the frequency domain [23]. Table 1 shows the relation between

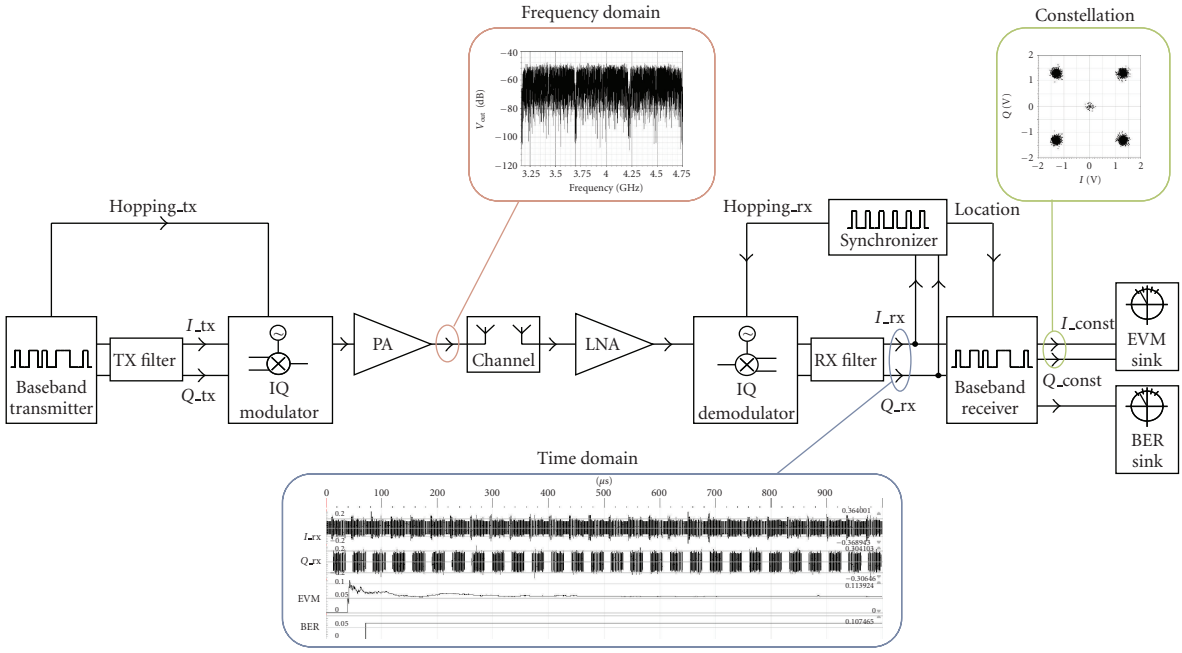


Figure 5: Abstracted view of the test bench and simulation results.

the chosen precision, the error introduced on the sampling time, and the consequent error Δf on the characteristic frequency of the signal.

For a precision of 1 picosecond, the frequency error relative to the signal bandwidth is 17 kHz over 528 MHz which is negligible. Furthermore, since the methodology makes use of only one solver, the same precision is applied to all signals in the system, thereby avoiding any synchronization problem.

The C code functions that describe the baseband algorithms process vectors representing the signal packet or frame without any notion of time. However, the capabilities of vector processing from within Verilog modules are limited with the consequence that the vectors have to be processed as a sequence of samples in the time domain. The unrolling and the packing of the vectors are preferably done directly in the wrapper using dedicated functions for controlling events and adding time information.

The vectors generated by the C code function of the baseband transmitter are unrolled in the wrapper to a sequence of samples transmitted with a specific time stamp as shown in Figure 4. The wrapper of the baseband receiver samples the signals and packets them into a vector to be processed by the C code function. This approach is very flexible since the timing information as well as the length of the vector can be adapted runtime according to the system parameters and are not statically defined as in the case of synchronous data flow solvers [24].

The data generated by the baseband transmitter are fed through the RF chain and are then sampled and recollected by the baseband receiver. For the correct functioning of the system, the sampling frequency of the signal is known and is the same for both the baseband transmitter and receiver.

Due to the fact that a discrete event solver is used for digital signal processing, the sampling rate is not a property of each signal as it is for example in the case of a synchronous data flow solver, where the rate is part of the scheduling process. The knowledge of the correct sampling time is essential for all signals that must be sampled by the successive blocks in the chain. The sampling time could have been retrieved with a specific function to recover the clock on the receiver side. In our case, to limit the complexity of the simulation, a clock is generated in the baseband transmitter and triggers the sampling of the signals in the successive blocks of the chain.

7. SIMULATION RESULTS

The test bench for the functional simulation of the UWB PHY and the performance results are described in this section.

The methodology is implemented using AMS Designer [7], the mixed signal simulator of Cadence.

Figure 5 shows an abstracted view of the test bench with the simulation results. The test bench combines the functional models of the baseband transmitter and receiver in C language, of the RF transceiver in Verilog-A or Verilog-AMS together with the performance metrics. The frequency spectrum is calculated using discrete fast Fourier transformations applied to the result of the transient analysis. The constellation diagram is drawn by plotting the quadrature output against the in-phase output of the baseband receiver. The presented simulation results are qualitative and the parameters of the functional models can be dimensioned

according to the functional requirement specifications of the UWB PHY.

The performance of the UWB system is evaluated in terms of metrics such as bit error rate (BER) and error vector magnitude (EVM). To evaluate the simulation performance of the complete UWB PHY, a transient analysis is run for 5 milliseconds which requires a simulation time of 3 hours. At the chosen rate of 110 Mbps, this corresponds to approximately 143 packets of data and 590304 payload bits. The number of processed bits is enough to evaluate the EVM. However, the accurate estimation of the BER requires a longer transient analysis.

To estimate the overhead introduced by the VPI, another transient analysis of 5 milliseconds is done using a test bench consisting of only the baseband models in C language. The total simulation time required by this setup is less than 3 minutes. From the difference between the two run times, it can be deduced that the contribution of the VPI to the simulation time is negligible.

The framework shows the possibility of implementing feedback loops. In the test bench, the synchronizer recovers the hopping sequence from the received data and controls the frequency of the local oscillator in the demodulator, showing the tight interaction between the baseband and the RF chain. This feedback loop is implemented without introducing artificial delays in the system description.

The analog part of the UWB system is simulated with a maximum time step around 25 picoseconds which corresponds to a simulation bandwidth of 20 GHz. Such a large simulation bandwidth not only allows the capture of the spectral regrowth but also facilitates the addition of interferers for coexistence studies.

8. CONCLUSIONS

The evolution of wireless communication standards has lead to systems in which analog, RF, and mixed signal functionalities have to be combined with digital signal processing algorithms, calibration, and correction loops. The functional simulation of such systems is a real challenge when using existing methodologies and simulation tools that have limited capabilities in handling mixed signal and mixed abstraction level designs.

This paper has presented a methodology that enables functional system verification in the time domain. The use of a mixed signal simulator and the Verilog procedural interface brings together analog and RF behavior in Verilog-AMS models with baseband algorithms in C language, enabling a true mixed signal and mixed abstraction level simulation environment. A single kernel framework is used avoiding any synchronization problem. The use of standardized languages and interface makes the methodology library and tool independent.

A demanding system, the ultrawideband physical layer, has been selected to evaluate the methodology. Analog behavioral models of the RF transceiver together with functional models of the baseband allow raising the level of design abstraction and coping with the system complexity. The test bench serves as a framework for complex mixed abstraction level simulations, where blocks at transistor level can be verified in the system context. Algorithm exploration for the calibration of impairments and compensation techniques in the RF transceiver is also possible using the same test bench. Feedback loops are simulated without any artificial delay introduced by the simulation framework.

The results shown in this paper confirm that the methodology can handle the complexity of modern multidisciplinary systems. A simulation platform is available from which the system architects, the designers of integrated circuits, and the algorithms can benefit, helping them to manage the complexity of next-generation wireless embedded systems.

REFERENCES

[1] NXP Semiconductors, UWB product literature, http://www.nxp.com/acrobat_download/literature/9397/75015695.pdf.

[2] "High Rate Ultra Wideband PHY and MAC standards," standard ECMA 368.

[3] S. Joeres and S. Heinen, "Mixed-mode and mixed-domain modelling and verification of radio frequency subsystems for SoC-applications," in *Proceedings of the IEEE International Behavioral Modeling and Simulation Workshop (BMAS '05)*, pp. 54–59, San Jose, Calif, USA, September 2005.

[4] Verilog-AMS Language Reference Manual. Analog & Mixed-Signal Extensions to Verilog HDL. Version 2.1, Accellera, http://www.accellera.org/.

[5] IEEE Standard VHDL Language Reference Manual IEEE Std 1076, 2000.

[6] Mentor Graphics, product data sheet, http://www.mentor.com/products/ic_nanometer_design/ms_circuit_simulation/advance_ms/upload/ADMS_Datasheet.pdf.

[7] Cadence Design Systems, product data sheet, http://www.cadence.com/datasheets/virtuoso_mmsim.pdf#page=7.

[8] Synopsys, product data sheet, http://www.synopsys.com/products/discoveryams/discoveryams_ds.pdf.

[9] U. Knochel, T. Markwirth, A. Hartung, R. Kakerow, and R. Atukula, "Verification of the RF subsystem within wireless LAN system level simulation," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, pp. 286–291, Munich, Germany, March 2003.

[10] S. Mu and M. Laisne, "Mixed-signal modeling using Simulink based-C," in *Proceedings of the IEEE International Behavioral Modeling and Simulation Workshop (BMAS '05)*, pp. 128–133, San Jose, Calif, USA, September 2005.

[11] A. Sayinta, G. Canverdi, M. Pauwels, A. Alshawa, and W. Dehaene, "A mixed abstraction level co-simulation case study using systemC for system on chip verification ," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, pp. 95–100, Munich, Germany, March 2003.

[12] G. Arnout, "C for system level design," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '99)*, pp. 384–386, Munich, Germany, March 1999.

[13] Agilet EEsoft EDA, http://eesof.tm.agilent.com/.

[14] The MathWorks, http://www.mathworks.com/.

[15] CoWare, http://www.coware.com/.

[16] U. Eichler, U. Knöchel, S. Altmann, W. Hartong, and J. Hartung, "Co-simulation of Matlab/Simulink with AMS Designer in System-on Chip Design," SNE16/2 2006.

[17] S. Joeres and S. Heinen, "Functional verification of radio frequency SoCs using mixed-mode and mixed-domain simulations," in *Proceedings of the IEEE International Behavioral Modeling and Simulation Workshop (BMAS '06)*, pp. 144–149, San Jose, Calif, USA, September 2006.

[18] C. Dawson, S. K. Pattanam, and D. Roberts, "The verilog procedural interface for the verilog hardware description language," in *Proceedings of the IEEE International Verilog HDL Conference (IVC '96)*, pp. 17–23, Santa Clara, Calif, USA, February 1996.

[19] F. Martinolle and A. Sherer, "A procedural language interface for VHDL and its typical applications," in *Proceedings of the International Verilog HDL Conference and VHDL International Users Forum (IVC/VIUF '98)*, pp. 32–38, Santa Clara, Calif, USA, March 1998.

[20] L. Chun, Y. Jun, G. Gugang, and S. Longxing, "Domain fault model and coverage metric for SoC verification," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '05)*, vol. 6, pp. 5662–5665, Kobe, Japan, May 2005.

[21] P. A. Riahi, Z. Navabi, and F. Lombardi, "A VPI-based combinational IP core module-based mixed level serial fault simulation and test generation methodology," in *Proceedings of the 12th Asian Test Symposium (ATS '03)*, pp. 274–277, Xian, China, November 2003.

[22] J. E. Chen, "Modeling RF Systems," http://www.designers-guide.org/.

[23] R. B. Staszewski and R. Staszewski, "VHDL simulation and modeling of an all-digital RF transmitter," in *Proceedings of the 5th International Workshop on System-on-Chip for Real-Time Applications (IWSOC '05)*, pp. 233–238, Banff, Canada, July 2005.

[24] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 738136, 16 pages
doi:10.1155/2008/738136

Research Article

Bridging MoCs in SystemC Specifications of Heterogeneous Systems

Markus Damm,[1] Jan Haase,[1] Christoph Grimm,[1] Fernando Herrera,[2] and Eugenio Villar[2]

[1] *Institute of Computer Technology, Vienna University of Technology, 1040 Vienna, Austria*
[2] *Microelectronics Engineering Group, TEISA Department, University of Cantabria, 39005 Santander, Spain*

Correspondence should be addressed to Jan Haase, haase@ict.tuwien.ac.at

Received 15 October 2007; Revised 21 February 2008; Accepted 14 May 2008

Recommended by Sandeep Shukla

In order to get an efficient specification and simulation of a heterogeneous system, the choice of an appropriate model of computation (MoC) for each system part is essential. The choice depends on the design domain (e.g., analogue or digital), and the suitable abstraction level used to specify and analyse the aspects considered to be important in each system part. In practice, MoC choice is implicitly made by selecting a suitable language and a simulation tool for each system part. This approach requires the connection of different languages and simulation tools when the specification and simulation of the system are considered as a whole. SystemC is able to support a more unified specification methodology and simulation environment for heterogeneous system, since it is extensible by libraries that support additional MoCs. A major requisite of these libraries is to provide means to connect system parts which are specified using different MoCs. However, these connection means usually do not provide enough flexibility to select and tune the right conversion semantic in a mixed-level specification, simulation, and refinement process. In this article, converter channels, a flexible approach for MoC connection within a SystemC environment consisting of three extensions, namely, SystemC-AMS, HetSC, and OSSS+R, are presented.

1. INTRODUCTION

Today's embedded systems mostly consist of digital hardware, analogue hardware and software, a circumstance often referred to as being heterogeneous. The description and simulation of such systems are usually associated to different teams with an expertise in a specific domain (e.g., signal processing, analogue hardware, digital hardware and software). Each team selects the most suitable language and simulation tool for its domain. For instance, digital hardware is described with an HDL and simulated with a cycle-based simulator, while an analogue part is described as a transistor-level schematic and simulated with solver-based tools.

The role of a system engineer in such a distributed work flow is to conceive the specification of the whole system and deliver it to domain specific teams. Often, the specification is a text, that is, written in English, while at other times, it is an abstract model written in a high-level programming language, such as C.

This approach has several drawbacks, where the major one is the late coupling of the different system parts. Specification flaws with respect to the overall system are detected quite late (after the development phase, instead of detecting them after the specification phase). In addition, the use of different languages results in additional difficulties. First, it involves the cumbersome task of coupling simulators, which mostly involves ad hoc solutions for respective simulator pairs. Moreover, it makes the overall system description more difficult to understand. Even if the different languages used have a similar syntax, they sometimes present subtle, but important semantical differences. For instance, the "signal" concept is employed in different metamodels, in RTOS interprocess communications, in the Esterel synchronous language and in VHDL, and is used for the communication of concurrent computations. However, the respective semantics of the "signal" happen to be very different. Thus, it is important to ensure that the system designer is able to deliver an unambiguous specification, where the meaning of each part is clear for all the agents involved in the design.

An alternative is to provide the system engineer with a methodology to produce an executable system-level specification whose constructs are expressive, general, and specific

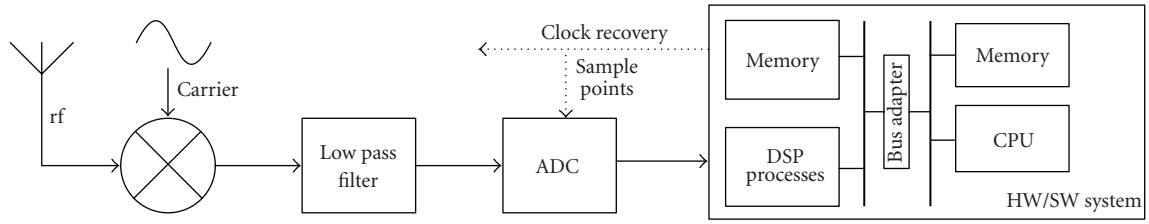


Figure 1: Example for an embedded analog/mixed-signal architecture: SW defined radio.

enough to be accepted and understandable by the different design communities. An executable specification enables a simulation-based process to discover specification flaws and validate the specification before implementing further design steps. A key point for such a methodology is to base it on a language which establishes a common and unambiguous basis in syntactical and semantical terms for specification and simulation tasks. At the same time, the design methodology must support gradual refinement of different parts of the specification. The effect of such refinement steps must be checkable globally, that is, within the framework of the whole system. As it will be seen later, this is related to the support of mixed-level simulation.

SystemC [1] is a system modelling language which is widely accepted and has features to become a common basis for system-level specification and simulation of embedded systems. SystemC is a C++ class library for system modelling and simulation with a discrete event (DE) simulation kernel. SystemC provides additional specification facilities to model time, concurrency, hierarchical partitioning, and hardware constructs like clocks and signal channels. In addition, the language provides facilities for extensibility. This has been recently exploited by several methodologies, based on extension libraries, like SystemC-AMS [2] and HetSC [3]. These libraries support additional MoCs in SystemC, which provide a system-level support for different application domains in a single system-level specification of a heterogeneous system.

Using these libraries enables, for instance, a faster simulation of analogue hardware and signal processing parts and a safer development of concurrent embedded software. Other SystemC extensions, like transaction-level modelling (TLM) [4], are focusing on the application of MoC concepts to support early and fast hardware platform modelling.

Figure 1 shows a software-defined radio (SDR) system as an example of an embedded mixed-signal application. Such a system samples the radio frequency input with a large sampling frequency. The signal demodulation then can be carried out by the digital hardware part with software.

The modelling and simulation of such a system give rise to using a number of different MoCs. Obviously, there is an analogue part to the left of the A/D converter (ADC), and a digital part, namely, the hardware/software (HW/SW) system. But the choice of MoCs for modelling within these two domains might depend on simulation requirements or the abstraction level of the different system parts. For example, the analogue part may be modelled with a signal flow-oriented MoC (like the SystemC-AMS-timed synchronous dataflow (T-SDF) MoC [5]), while the *functionality* of the whole HW/SW system may be modelled using the Kahn process network (KPN) MoC first, abstracting its structure as indicated in Figure 1.

Later refinement steps might require an electrical network model for the low-pass filter, while for the HW/SW system, a structural model is provided with the bus communication abstracted using TLM concepts. A subsequent refinement step then uses a clocked synchronous model for bus communication, while the digital DSP part may be modelled using the T-SDF MoC for simulation speed purposes. This example indicates that the usage of various MoCs in modern system design is natural, and indeed inevitable if a profound theoretical basis is desired. This, in turn, implies the need for conversion means between system parts being modelled using different MoCs. These conversion means should be easy to handle.

1.1. MoC-conversion

While SystemC extensions enable the designer to model and simulate heterogeneous systems with custom MoCs for distinct system parts, their usage also requires conversion means between system parts modelled using different MoCs. These conversion means have to adapt time, communication, and computation domains of the respective system parts, with time adaptation being of capital importance. By abuse of language, we will refer to this conversion process as *MoC-conversion*, in accordance with [6], although one might argue that terms like *MoC-bridging* or *MoC-composition* are more appropriate.

In some cases, the required conversion is natural and explicit, for example, when considering a system consisting of digital and analogue hardwares. Such a system will contain A/D- and/or D/A-converters from the specification level down to the implementation level. Concepts like *mixed-level simulation* [7] give rise to more artificial MoC conversion problems. The idea of mixed-level simulation is to simulate systems such that different parts are specified at different levels of abstraction. The benefit of this approach is the combination of the speed of the system-level simulation with the accuracy of simulation on more detailed levels. The drawback is that every time a system part is refined to a lower abstraction level, the respective MoC will often change

too, while the other parts remain at a higher abstraction level with their MoC unchanged. Therefore, MoC converters are needed here for the sole purpose of modelling and simulation.

MoC converters are usually "static" in the sense that they perform a single adaptation, for example, $MoC_1 \rightarrow MoC_2$. For example, SystemC-AMS provides converter ports to connect T-SDF-modules to discrete event signals. However, the support provided by these static converters can be improved to speed up refinement steps and design space exploration. Each refinement step in a mixed-level design approach may involve a different static MoC converter. This can require a systematic, error-prone, and time-consuming replacement of such static converters.

This article presents an approach to automate the usage of MoC (and also data type) conversion with a SystemC channel called *converter channel*. Converter channels detect the MoCs of the system parts they are connected to via the interface types of the respective ports, and provide automatically the conversion means needed. If the conversion semantic is not obvious, options are provided for the designer to steer the behaviour of the converter channel in certain cases. Moreover, since the conversion is hidden within the channel, it does not interfere with the reuse of partial design blocks.

The rest of the article is organized as follows: Section 2 gives an overview of the related work. Section 3 introduces the two SystemC libraries which the converter channels are based on, namely, HetSC and SystemC-AMS. In Section 4, we show how a converter channel is used. Section 5 treats the internal structure of converter channels, and gives details on how two specific MoC conversion cases are handled. In Section 6, we give an application example. We conclude in Section 7.

2. RELATED WORK

This article focusses on the improvement of MoC connections, a key issue in heterogeneous specification. The work done on heterogeneous specification will be reviewed in the following sections. It will situate this work and will also show its basis.

2.1. Metamodels

Metamodels are an important part of the theoretical study of MoCs. They provide general and formal ways to study and compare different MoCs. This is key to understand the properties of each MoC, as well as the interactions which arise when two or more MoCs are combined in the same specification.

In [8], any concurrent specification is abstracted as a set of *processes* connected by means of *signals*. A *signal* is understood as a set of *events*, while an *event* is a value-tag pair. Neither an order relationship is assumed among the events (although it can be established as a function of the event tags), nor is there any additional implication in terms of communication semantic. Finally, a *process* is defined as a relationship among input and output *signals* of the *process*. This view of a process is quite different from that of a process being a sequence of statements able to change a set of state variables related to it.

ForSyDe (formal system design) [9] is a metamodel also based on *process*, *signal*, and *event* concepts. However, it presents slight differences with respect to the metamodel in [8]. A signal is a sequence of *events*, where an *event* is just a value. Despite there is no tag associated to a ForSyDe event, the definition of a signal as a sequence provides a notion of strict order among the events belonging to it. In addition, ForSyDe defines three types of events, which provide the capability to represent untimed, synchronous, and timed models. A major difference to [8] is that a ForSyDe process is defined in an introspective way, by means of process constructors, which take as input the functions which relate the input and output signals. A strength of ForSyDe is a set of transformation rules, which enable a safe formal procedure for the MoC refinement process involved in the design process, when the specification is refined into the implementation.

Although metamodels provide a formal basis for a heterogeneous specification methodology, a metamodel is not a methodology itself. ForSyDe, however, is also based on a Haskell library [10], such that ForSyDe models can actually be implemented and simulated. A handicap is that Haskell is not quite extended among the design community.

There are several approaches to provide heterogeneous specification methodologies. Two main approaches can be distinguished. On one hand, the extension of hardware description languages (HDLs), mainly for analogue extensions, on the other hand, the extension of high-level languages like UML, Java, or C++. These proposals will be reviewed in the following sections.

2.2. Extension of HDLs: Verilog-AMS and VHDL-AMS

The mixed signal hardware description language Verilog-AMS [11] incorporates two different MoCs for simulating analogue and digital hardware, respectively. While digital simulation is handled by a DE simulation kernel, a differential equation kernel is engaged with the simulation in the analogue domain. *Disciplines* can be defined to be associated to these domains, for example, an analogue discipline for electrical signals or digital disciplines for binary or multivalued logic. To connect modules belonging to different disciplines, Verilog-AMS provides the features of *connect modules* and *connect modes* [12]. Connect modules must be implemented by the designer and define the interaction between modules of different disciplines. Apart from signal conversion, connect modules can also model subtle behaviour like the influence of a capacitor to the delay of a digital signal. With connect modes, the designer steers the automatic insertion of these connect modules.

The VHDL-AMS language [13] is an extension of the IEEE 1076 (VHDL) standard that supports description and simulation of analogue, digital, and mixed-signal circuits and systems. A basic principle is not to rely on a specific algorithm, but on a mathematical foundation to solve the implicit or explicit differential algebraic equations (DAE)

which describe the analogue part of the system specification. Models must not depend on the time steps taken by the analogue solver. VHDL-AMS adds two new object types to VHDL: the *quantity* and the *terminal*. Both can either be a local or an interface object, and they are used in conjunction with basic VHDL primitives, like *entity* and *architecture*. Specifically, input and output ports can be declared either as terminals or as quantities. Quantities are floating point scalars, which are the unknowns of the DAEs solved by VHDL-AMS. Quantity associations create a signal flow model. Terminal associations of the same *nature* can be used to create an analogue net list. A nature restricts the association of terminals to those of the same nature. A nature also relates *across aspects*, which are quantities which represent effort like effects (voltage, pressure, etc.), with *through aspects*, which represent flow-like effects (current, fluid flow rate, etc.). An example of nature is the *electrical nature*, which relates voltage quantity with current quantity. These basic elements let the user specify implicit DAEs. A set of explicit concurrent statements enables the specification of explicit DAEs. Among them, the simultaneous procedural statement allows the formulation of the DAE as inline sequential code.

Analogue extensions of HDLs provide support for the specification of mixed signal systems, combining digital, and analogue domains. However, they lack support for more abstract MoCs, and are able to consider abstract models in the early design phases and for embedded software, an important part of embedded systems. We now review specification frameworks based on extensions of high-level languages, which cover these deficiencies better.

2.3. Extensions of high-level languages

The lack of a unified system specification language is one of the main obstacles bedevilling SoC designers [14]. A common specification language is a major aid in generating a heterogeneous specification methodology. It states a common and unambiguous syntax and semantic for the specification facilities. This improves the understanding between the different design areas and enables exchanging the unambiguous models, independent of the tools used by each part for graphical capture, verification, performance analysis, synthesis, that is. Therefore, extensions of programming languages like UML, Java, or C++ have been proposed for system modelling and simulation.

2.3.1. Ptolemy II

Ptolemy II [15] (the latest evolution of the Ptolemy project) is one of the first and most important approaches to provide a unified framework for the specification and simulation of heterogeneous systems. A Ptolemy specification consists of a set of components called *actors*. *Actors* communicate among them by means of *ports*. *Actors* can be either composed out of a set of other actors or primitive. In the latter, the actor has a specific functionality described in Java. The Java functionality is distributed in a set of internal methods (*initialise*, *prefire*, *fire*, *postfire*, etc.), which are callbacks that execute functional code and perform read and write accesses to the actor ports at different times of the simulation. These callbacks are governed by a Ptolemy *director*. A *director* is associated to an *actor* and defines its *domain*, that is, the MoC. The callback structure and a graphical capture environment called *Vergil* provides a kind of common specification framework to Ptolemy, while Java has more an implementation role.

Heterogeneity is handled through hierarchisation, in the sense that actors on the same hierarchy level obey the same *director*, thus the same MoC. Each specification instantiates its own hierarchy of directors, which coordinates actor executions. This helps MoC connection, since the order and time position in which each token is transferred across the domains is explicitly controlled by the directors' hierarchy. Because of this, in general, no special MoC interface infrastructure to connect actors of different domains is necessary, apart from port communication. In addition, Ptolemy introduces the *ModalModel* for models which let specify different modes of operation for a model under any of the supported MoCs. By means of a discrete FSM, the different mode transitions which the model can traverse is specified. Each mode has a *refinement*, under the specific model embedded in this mode FSM. Then, *Hibrid Systems* embed continuous time models in a mode FSM. In any case, Ptolemy provides a hierarchical approach to heterogeneity.

2.3.2. SysML

SysML (systems modelling language) [16] is a domain-specific modelling language for system engineering applications. It supports the specification of systems which may include hardware, software, information, processes, personnel, and facilities. The main idea is that UML is well suited for software modelling, but too comprehensive and imprecise for domain-specific applications which require nonsoftware components. Therefore, SysML is constituted as a *profile* which reuses a subset of UML 2.0. At the same time, SysML also extends UML 2.0 providing diagrams for the capture of requirements and parametric constraints. These capabilities enable requirements engineering and performance analysis, two major activities in system design.

SysML was originally developed as an open source specification project. In 2005, the OMG (bject management group) derived the OMG SysML [17], which is defined as SysML, highlighting it as a graphical language. Graphical capabilities of SysML for the specification capture are endorsed by a wide set of modelling tools supporting UML 2.x and the SysML profile. Thus, SysML provides a common, rich, and graphical method to model systems under specific design domains. However, SysML, as other UML profiles must rely on an implementation language to be executable or simulatable. For instance, in [18], SysML is used for the specification of analogue systems. That is, SysML is used to model the dynamics of continuous systems by means of DAEs. However, since UML lacks analogue solvers, the solution adopted is to translate SysML constructs to Modelica language constructs.

2.3.3. Metropolis

Metropolis [19] is a specification methodology which is (like Ptolemy) based on Java. It also targets simulation, verification, and synthesis (mainly of software). Metropolis supports heterogeneity in two levels. In the lower level, it defines a metamodel language which provides a class-based infrastructure. The classes have a well-defined semantic, but are general enough to support existing MoCs and new ones layered in an upper level.

In Metropolis, there are basic specification elements, like the *process*, the *port*, and the *interface*, which are similar to other specification methodologies like SystemC. *Processes* are atomic computation elements, defined as sequences of events mapped to threads. The *port* is the unique mean of a process to communicate with other processes. *Interfaces* are a set of communication methods. Other specification facilities, for example, *media*, *quantity manager*, and *state media*, are specific of Metropolis. *Media* are the primitives for communicating processes. They connect processes through *ports*. *Quantity managers* fix constraints whether processes must be scheduled. *Quantity managers* communicate among themselves by means of *state media*. The metamodelling character of the lower level of Metropolis is confirmed by the ability to combine executive parts with declarative parts. Declarative parts enable the specification of nonfunctional constraints.

Metropolis is able to support heterogeneous design building specific models over a general framework. This is mainly achieved by means of the different implementations of *media*, whose semantic mostly defines the MoCs employed. This is a major distinction with respect to Ptolemy, where MoC semantic mostly relies on the *director* class and where *actors* connections play a more secondary role. Strengths of Metropolis are its high expressivity and its formal framework, which enables the connection of synthesis and verification tools.

Regarding MoC connection, the Metropolis metamodel defines a core element called *adapter* or *wrapper*. It is modelled as a process in charge of adapting the process which contains the actual functionality to the medium this *functional* process is connected to. This adaptor determines the firing condition of the functional process and the interaction with the connected media. Therefore, when processes are refined and different MoC connections are involved, this requires only changing the adapter (the functional processes are not affected). In addition, *Quantity managers* can somehow be employed to control the execution semantic of the different components of the specification, thus to ensure a coherent coordination of their simulation, as it is done in Ptolemy by means of *directors*.

2.4. SystemC extensions for heterogeneous specification

Several reasons lead research community to propose C++ based extensions for heterogeneous system-level specification, which make use of the object-orientated features of C++. Executable specifications using such extensions are compilable and do not rely on a virtual machine, which can lower the simulation performance, a key aspect in system-level design activities. In addition, C++ syntax is familiar to many embedded system and HDL designers.

Talking about C++ extensions for system-level specifications has become almost synonymous to talking about SystemC extensions. SystemC is an IEEE standard language that has started to play a role as unifying system-level language for embedded system design. The SystemC language reference manual [20] states a syntax and an unambiguous semantic for the language constructs, which is a strong basis for SystemC-based methodologies supporting heterogeneous specification.

2.4.1. SystemC-H

SystemC-H [21] is a methodology that proposes a general extension of the SystemC kernel for the support of different MoCs. The extension would include a solver for each supported MoC. The current scope of the SystemC-H library covers the SDF and communicating sequential processes (CSPs) untimed MoCs. For instance, SystemC-H provides a solver for static scheduling of SDF graphs which enables schedulability analysis and provides a 75% speedup with respect to the DE MoC [21]. Another result of this work is that the addition of a specific solver is not always worthwhile, since the simulation speedup for some abstract MoCs can be negligible [22]. In addition, similar speedups were reported for the dynamic approach to SDF for large-grain SDF specifications [3]. Finally, the combination of MoCs, can also spoil the speedup gained by using a specific MoC. For instance, the speedup decreases to 13% for a mixed DE-SDF example [21]. A major drawback of SystemC-H is that it is implemented by modifying the SystemC standard library.

2.4.2. Analogue extensions of SystemC

SystemC-A [23] proposes a general approach to provide analogue extensions in SystemC which is able to handle a wide range of nonlinear dynamic systems. SystemC-A is a superset of SystemC which provides support for several abstraction levels, with a special focus on a higher abstraction level. However, it also supports circuit-level descriptions. SystemC-A proposes a general-purpose analogue solver coupled with the DE kernel of SystemC by means of pessimistic (lock-step) synchronisation. However, this synchronisation requires the modification of the SystemC kernel.

The SystemC-WMS [24] library provides basic electronic components and blocks for analogue macromodelling, such that the specification of analogue models can combine parts at low- and high-abstraction levels. Analogue blocks can communicate by exchanging energy waves using *wavechannel* interfaces. Thus the methodology enables the definition of a standard analogue interface. In [24], it is also claimed that SystemC-WMS can support simple types of nonlinear DAEs.

HetSC and SystemC-AMS are proposals for extending SystemC for heterogeneous specification which are directly

involved in our work. They will be reviewed in detail in the following chapter.

3. HetSC AND SystemC-AMS

The task of using HetSC together with SystemC-AMS for system modelling and simulation came up due to the project ANDRES (**AN**alysis and **D**esign of runtime **RE**configurable heterogeneous **S**ystems [25]). ANDRES explores design methodologies for heterogeneous systems which are able to adapt themselves during runtime to changing user requirements as well as varying environmental constraints. The project encompasses the development of

(i) a theoretical framework for the specification of such adaptive heterogeneous systems (AHESs);

(ii) a SystemC-based modelling framework where the different domains are represented by specific MoCs;

(iii) means to automatically synthesise the components and interfaces of AHES regarding the digital hardware and the software.

The work presented here is an integral part of ANDRES. While the key concern of ANDRES is the consideration of adaptivity in all the domains involved, it is obvious that the integration of the MoCs and methodologies involved in the project is essential. The SystemC extensions used are the following.

(i) *SystemC-AMS* [2] targets modelling and simulation of analogue and signal processing systems. It provides the MoCs (timed) SDF and continuous time (with respect to linear electrical networks (CT-NET)).

(ii) *HetSC* [3] is intended to model systems using different MoCs (e.g., Kahn process networks (KPN) and synchronous reactive (SR) systems), and also provides an entry point for software synthesis.

(iii) *OSSS+R* [26] is a library for object-oriented modelling of run-time reconfigurable hardware, which will provide direct hardware synthesis capabilities targeting dynamically reconfigurable FPGAs.

The difficulty lies in getting the three libraries to work together with respect to the different MoCs involved. Since the modules in OSSS+R are basically clocked synchronous, which is a MoC provided by SystemC directly via its discrete event (DE) kernel, this is a challenge which mainly concerns HetSC and SystemC-AMS.

3.1. HetSC

HetSC [3] is a methodology for enabling heterogeneous specifications of complex embedded systems in SystemC. MoCs supported include untimed MoCs, such as KPN, its bounded FIFO version bounded Kahn process networks (BKPNs), communicating sequential processes (CSPs), and synchronous dataflow (SDF). Synchronous MoCs, such as synchronous reactive (SR), clocked synchronous (CS), and

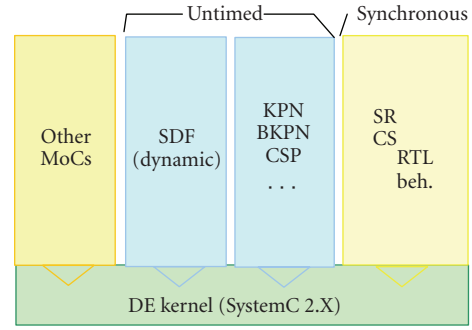


Figure 2: HetSC MoCs over the DE strict-time kernel of SystemC.

the timed MoCs already supported in SystemC are also included (see Figure 2). HetSC aims at a complete system-level HW/SW codesign flow. Indeed, the methodology has been used together with other system-level profiling and software generation methodologies [27].

The HetSC methodology defines a set of specification rules and coding guidelines for each specific MoC, which makes the designer task more systematic. The support of some specific MoCs requires new specification facilities providing the specific semantic content and abstraction level required by the corresponding MoCs. The HetSC library, associated with the HetSC methodology, provides this set of facilities to cover the deficiencies of the SystemC core language for heterogeneous specification. In addition, some facilities of the HetSC library help to detect and locate MoC rule violations and assist the debugging of concurrent specifications. One of the main contributions of HetSC is the efficient support of abstract MoCs (untimed and synchronous). This is because they are directly supported over the underlying discrete event (DE) strict-time simulation kernel of SystemC. New abstract MoCs do not require additional solvers since the new MoC semantic is embedded in the implementation of the new specification facilities (usually channels) related to the abstract MoC. In [3], a simulation speedup similar to that of kernel extension was reported for large grain specifications. It was argued too that once analogue parts are connected to the model, usually more critical in terms of simulation time, the consideration of Amdahl's law, can make almost undistinguishable the differences between the speedups got by the different implementations of abstract MoCs, independently on whether such implementations extends the kernel, like in [21], or to the contrary, it is layered over the strict-time DE kernel of SystemC, like HetSC does. HetSC provides the strategical advantage of not requiring the modification of the standard SystemC kernel, since the new features are provided by the HetSC library, what makes it a more decoupled approach.

3.2. SystemC-AMS

SystemC-AMS [2] is a specification methodology developed by the open SystemC Initiative (OSCI) SystemC-AMS working group. The main goal is to extend the HW/SW-oriented SystemC towards a framework that supports

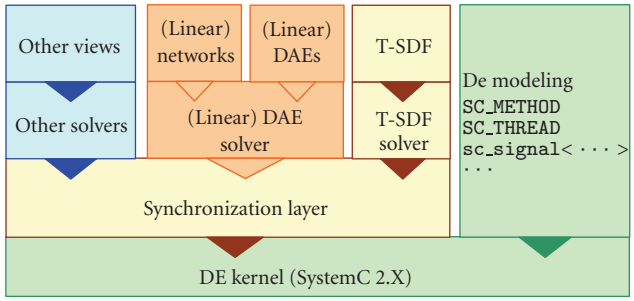


Figure 3: SystemC-AMS layers over the strict time DE kernel of SystemC.

functional modelling, architecture exploration, system integration, verification and virtual prototyping of embedded Aanalogue/mixed-signal (E-AMS) systems. The most important MoCs (in addition to the DE-MoC) required for this purpose are

(a) continuous time electrical networks (CT-NET);

(b) continuous time differential equations/transfer functions (CT);

(c) timed synchronous data flow (T-SDF).

Due to the interaction between the SDF and the other timed MoCs, SystemC-AMS attaches time semantics to SDF by settings fixed time steps to SDF cluster execution times. We therefore refer to the SystemC-AMS SDF MoC as "timed SDF" (T-SDF). For modelling asynchronous systems, time steps can also be triggered by events.

A major challenge is gaining sufficient simulation performance while accurately modelling the architecture's behaviour at the same time. At least for electrical networks, no mature and dependable approaches are available that allow us to stay in the SystemC 2.2 framework without kernel extensions. Simulation of electrical networks requires a structural analysis, setup of equation systems, and numerical methods for solving them. For T-SDF, measurements indicate that the use of extended kernel capabilities provides a factor 4 speedup for DSP functions frequently found in E-AMS such as FIR filters [28]. Therefore, SystemC-AMS provides kernel extensions for CT-NET, CT, and also T-SDF simulation.

The benefit of using T-SDF regarding simulation speed is twofold. Since it is SDF, it is possible to compute a static schedule for the T-SDF processes before the simulation starts (like in the current SystemC-AMS prototype), such that the scheduling overhead during the simulation is minimised. Additionally, using large data rates leads to schedules where certain processes might be computed repeatedly in a row, such that the number of context switches reduces also.

To synchronise the kernel extensions with the SystemC DE-kernel, SystemC-AMS uses T-SDF with discrete time steps and/or controlled time steps. This has the advantage that all modules are executed in signal flow's direction. Partitions of a model, simulated by one kernel extension, are encapsulated in a SDF module. The partition communicates with other components that use other kernel extensions only via directed (T-SDF) signals. For convenience, however, electrical networks can also interface DE directly.

The implementation of SystemC-AMS is organised in three layers. The top layer (view layer) includes the classes visible directly by a designer, and is used directly for specifying models. It provides, for example, an SDF-module class together with SDF-signals and port. The middle solver layer has kernel extensions that are used by the view layer. To couple different solvers, SDF with constant time and with user defined synchronisation events is used in the solver layer. Figure 3 shows the overall organisation of the SystemC-AMS extensions.

3.3. Connecting HetSC and SystemC-AMS

In [29] the connection of SystemC-AMS and HetSC was explored. It was shown, that both methodologies can collaborate to support a wide spectrum of MoCs. Moreover, the collaboration of these methodologies provides an efficient balance between MoCs directly supported over the DE strict-time kernel, and MoCs relying on additional synchronisation and solver layers. The idea is that specific solvers are provided only for a set of MoCs where the simulation speed up is significant. In this approach, this set corresponds to analogue MoCs where the simulation speed ups can be of several orders of magnitude, while untimed and synchronous MoCs can be satisfactorily supported directly over the SystemC kernel. The exception would be fine grain SDF specifications, where the speed up of a static SDF compared to a dynamic SDF could be significant. Thus, in these cases, the static scheduling provided by the T-SDF solver of SystemC-AMS should be favoured.

In [29] the ways in which HetSC and SystemC-AMS enable MoC connection was also explored. HetSC provides the concept of *border processes* and *border channels* for the connection of the various MoCs. In this way, common SystemC elements are employed in MoC interfaces. Border processes are similar to Metropolis adapters in that it requires from the user to explicitly write the adaptation code. In contrast, border channels hide this adaptation code. Border channels make a syntactical adaptation. For instance, on one side they can offer a FIFO like interface, while on the other side, they offer a rendezvous-like interface. Moreover, border channels can also adapt different semantics of connected MoCs. HetSC border channels mainly focus on the adaptations performed in the time domain. SystemC-AMS also provides similar ways for MoC connection. It provides converter ports and facilities to enable communication between different MoCs (i.e., DE with T-SDF, T-SDF with CT-NET, etc.). In other cases, MoCs, such as transfer function models, are actually embedded in T-SDF modules.

For SystemC-AMS, *polymorphic signals* [30] were developed to connect modules modelled under different MoCs (CT-NET, T-SDF or SystemC-DE). Like the HetSC border channels, they have to do a syntactical and semantical adaptation. An important benefit of polymorphic signals is that they are able to select the right MoC connection automatically. This is done before the simulation starts

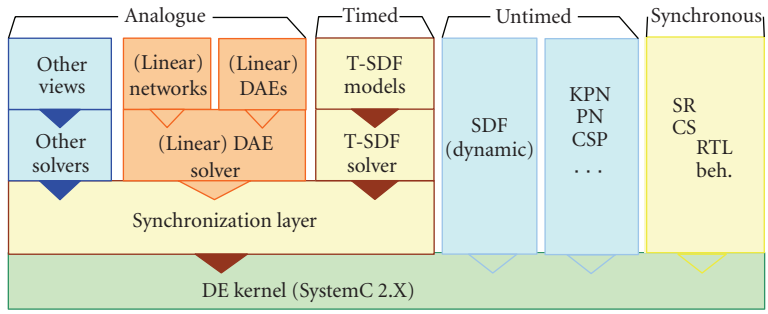


Figure 4: Integration of SystemC-AMS and HetSC.

(at *elaboration time*). The polymorphic signal detects the MoCs of the modules it is connected to and provides the appropriate conversion means without the designer's assistance who only sets different options.

4. CONVERTER CHANNELS

Converter channels, first introduced in [29], are meant to unify the approaches of polymorphic signals and border channels, while providing adaptation in the data type domain. Converter channels can connect modules specified using different MoCs, and automatically adapt their different handling in the semantic of time, communication, and data types. In this way, converter channels provide an advanced facility for the automatic syntactical and semantical connection in heterogeneous specification, refinement and design exploration methodologies based on SystemC.

Before we describe how a converter channel works, we demonstrate its use, that is, how they appear to the designer. The internal structure and conversion semantics will be covered in the next chapter. A converter channel is instantiated with up to six template parameters:

```
converterchannel < DT_W, DT_R1,...,DT_R5 > name;
```
(1)

the first parameter, `DT_W`, is mandatory and denotes the data type of the writing port the signal is going to be attached to; it can also be used as a data type of a reading port. The parameters `DT_R1` to `DT_R5` are optional and denote additional data types on the reading side.

The choice of the number of additional reading side data types to be five is somewhat arbitrary, but since the use of an unlimited number of template parameters in C++ is not possible, a number had to be fixed. It is reasonable to assume five additional data types to be more than sufficient for any reasonable application. However, if the compilation overhead proves to be to large, we might reduce this number.

The connection of the converter channel to module ports has to be done by named mapping and works like in Algorithm 1, where a T-SDF-module producing double values is connected to a T-SDF-module reading integer values and an ordinary SystemC-module (DE) reading double values (see also Figure 5).

As Algorithm 1 shows, the usage of a converter channel differs a little from the usage of an `sc_signal`

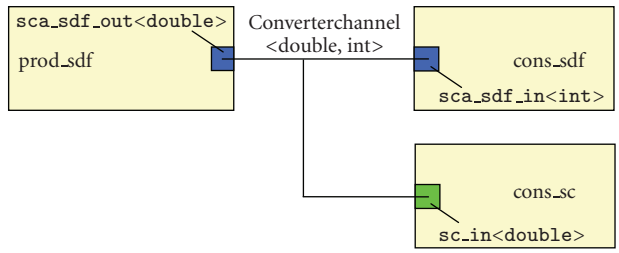


Figure 5: Illustration for Algorithm 1.

```
converterchannel < double, int > sig;

producer_sdf_double prod_sdf ("prodsdf");
prod_sdf.out (sig.source_sdf());

consumer_sc_double cons_sc("conssc");
cons_sc.in (sig);

consumer_sdf_int cons_sdf ("conssdf");
cons_sdf.in (sig);
```

Algorithm 1: Connecting a converter channel.

regarding the connection to the writing port, which contains a declaration of the MoC used on the writing side: instead of `prodsdf.out (sig)`, the code line `prodsdf.out (sig.source_MOC())` has to be used, where MOC denotes the MoC on the writing side. In contrast, the connection to ports of reading modules works as usual; in particular there is no need to declare the MoC of the reading module.

The need for MoC-declaration on the writing side has technical reasons. In principal, MoC and sense of the signal flow of a port (i.e. in- or output port) can be recognised automatically by the port's interface type. Unfortunately, for the `sca_sdf_port <>` class of SystemC-AMS, both the in- and output ports happen to implement the same interface. Therefore, we can recognise the MoC here, but not the signal flow sense. Since there is only one writer for each converter channel, using a MoC-specific binding method for the writing side has the least coding overhead. Additionally,

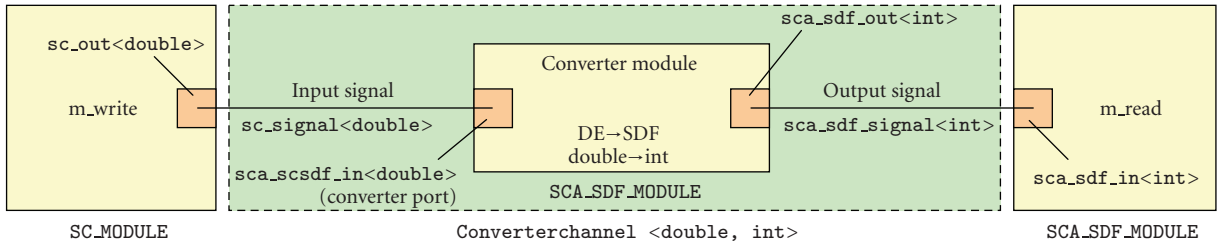


Figure 6: Example of the internal structure of a converter channel.

this approach offers the opportunity to pass additional MoC specific parameters to the converter channel using the binding method.

The designer has to make manual changes at a converter channel only if

(i) the MoC of the writing module is changed; then the connection of the converter channel to the port has to be changed, for example from

```
module.out (signal.source_sdf ())
```
(2)

to

```
module.out(signal.source_sc ())
```
(3)

(`sc` represents the standard SystemC DE MoC).

(ii) the data type of a port of a reading module is changed and is not already a template parameter in the instantiation of the converter channel; then it has to be added. So, for example, the code of the instantiation could change to

```
converterchannel < double, int, sc_bv < 8 >> sig;
```
(4)

furthermore, it is possible to set options on the converter channel, for example, to control the data type conversion semantics. If, for example, the writing side of a converter channel is of type `double`, with a `sc_uint < N >` port on the reading side, this situation has the potential for information loss if `N` is too small or if the writing side provides negative values. For such a case, the converter channel offers a method

```
sig.setRangeScaling (min, max)
```
(5)

to adapt the value range of the writing side to those of the reading side. For applying this function, the designer has to know (or has to have at least an idea of) the range of the input signal. Therefore, if the values of the writing side turn out to exceed the interval [min, max] during the simulation, warnings are produced.

This example also illustrates the potential risks of the automated data type conversion capabilities of a converter channel. We will not, however, go into detail on data type conversion issues in this article. More on this can be found in [31].

5. INTERNAL STRUCTURE AND OPERATION

In this section, we provide details on how the code described above is processed, that is, handled internally in terms of signals and converter modules. Let us assume that a converter channel connects two modules `m_write` and `m_read` having a port out and in respectively. Four cases can occur:

(1) The MoCs of the modules as well as the data types of the ports mismatch.

(2) The MoCs of the modules are different, but the data types are the same.

(3) The MoCs of the modules match, but the data types of the ports do not.

(4) Both MoCs and port data types match.

In case 1, the converter channel instantiates two appropriate signals (the input and the output signal), which then are connected to the ports of `m_write` and `m_read` respectively. These, in turn, are connected to an appropriate converter module. An example is shown in Figure 6: `m_write` is a SystemC DE module having a port `out` of type `sc_out < double >` and `m_read` is a SystemC-AMS T-SDF module having a port `in` of type `sca_sdf_in < int >`.

To achieve conversion between SystemC-DE modules and SystemC-AMS T-SDF modules, the SystemC-AMS library provides `sca_scsdf_in` and `sca_scsdf_out` ports, which can be used within T-SDF-modules to connect to DE-signals (i.e., `sc_signals`). Therefore, a converter module from DE to T-SDF can be realised by a T-SDF-module having an `sca_scsdf_in` input and an `sca_sdf_out` output. Then it can be connected to `m_write` and `m_read` with an `sc_signal` and `sca_sdf_signal` respectively. We will discuss the related conversion semantics shortly.

After reading the input signal, the value is converted to the desired target data type and written to the output of the converter module. The data type conversion function here is inherited by a special data type conversion class which has templated specialisations for each data type pair. Hence, the conversion modules are implemented using two separate levels: MoC conversion and data type conversion.

In case 2, the procedure is similar; there is no need for data type conversion, and the converter module just passes the value of the input signal to the output.

Case 3 is also fairly similar, with the difference that all ports and signals used are of the same type with respect to

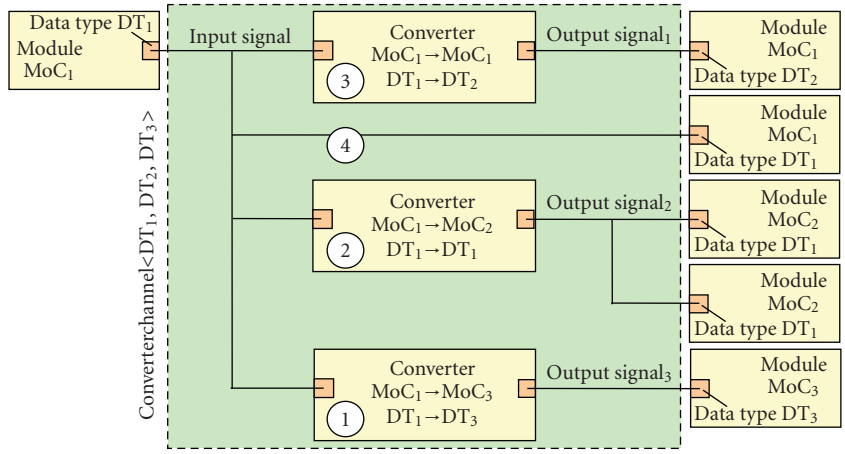


Figure 7: Internal structure of a converter channel connected to multiple reading modules.

the MoC; in particular the converter module belongs to the same MoC as the two modules to be connected. As in case 1, the input signal is read and the converted value is written to the internal output signal.

Obviously, the simplest case is case 4, where there is no need for any conversion at all. Here, the converter channel simply generates only the appropriate input signal, and connects it to `m_write` and `m_read` directly without a converter module in between.

In general, a converter channel will be connected to several readers. Here, apart from the input signal, several converters and output signals are created. Each converter module is then connected to the input signal and to its own output signal. Each reading module is connected to the appropriate output signal. If a reading module happens to agree to the writing module regarding MoC and data type, it is directly connected to the input signal. Figure 7 shows an example (the numbers refer to the four cases stated above). The capability of the converter channel to have readers with several data types is also an extension of the concept of polymorphic signals [30], where data type conversion was also included, but the reading modules had to share the same data type.

In the next section, we show how MoC conversion is handled in the case of DE↔T-SDF and KPN/BKPN↔T-SDF conversion.

5.1. DE↔T-SDF conversion

The conversion between T-SDF of SystemC-AMS and the SystemC DE MoC is basically straightforward, due to the strict time semantics of T-SDF. SystemC-AMS already provides conversion facilities for this, which are used by the converter channels, while also resolving potential semantical issues.

In the T-SDF→DE case, the converter channel instantiates an internal converter module which is connected to the writing side via an `sca_sdf_signal < DT_W >`, and which makes use of the SystemC-AMS `sca_scsdf_out <>` converter port to connect to the reading side via an internal `sc_signal < T >`.

A token which is written to the internal `sca_sdf_signal < DT_W >` by the writing side at SystemC-AMS time t is converted such that the internal `sc_signal < T >` holds the value of that token from SystemC time t on. Note that there is a value change event on the reading side *only* if there is an actual value change of one token to the next one, which might be a semantical problem in certain models, when the DE side is supposed to read *every* token. For these cases, the converter channel offers a clock signal running with the pace of the T-SDF side, which can be accessed using the method `sdf_clock ()`.

With respect to the previous case, the DE→T-SDF case is exactly mirrored regarding the use of internal signals and converter ports (see Figure 6). The conversion semantics is such that a value which is written to the internal `sc_signal < DT_W >` by the writing side constitutes a *current* value for the signal. This current value is then sampled by the T-SDF side with a frequency determined by its sampling period.

There is, however, a potential for loss of information. If the DE side changes the value twice or more in between two sampling instances by the T-SDF side, these interim values will be lost. Therefore, the converter channel provides warnings in this case with the help an internal DE-module which is sensitive to the value changes of the internal `sc_signal < DT_W >`. At each sampling instance, the internal converter module asks this module how many value changes occurred since the last sampling instance. If this value exceeds one, it raises a warning or a runtime error respectively, depending on the designer's choice. This behaviour can also be switched off completely.

5.2. KPN/BKPN↔T-SDF conversion

In [32], several types of MoC connections were distinguished with regard to the time domain. The basic observation is that when two MoCs are connected which handle time at a

different detail level, there is an effect of time information injection from the more detailed MoC to the less detailed MoC. The way by which time is handled is often related to other aspects associated to communication semantics. For instance, it is common to associate consuming reads and nondestructive writes to untimed MoCs, while destructive write and nonconsuming reads semantic of HDL signals (e.g., the `sc_signal` channel semantic in SystemC) is suitable for timed-clocked MoCs.

In previous works, semantical issues dealing with untimed-untimed [32] and untimed-synchronous [33] MoC connection have been addressed. We will consider a first approach to the untimed-timed and the timed-timed MoC connections, where the untimed-timed case refers to the connection of untimed Kahn process network (KPN) and bounded Kahn process network (BKPN) MoCs of HetSC with the timed synchronous dataflow MoC (T-SDF) of SystemC-AMS. The timed approach of T-SDF can actually be considered as a discrete time MoC, since each cluster execution has a specific SystemC time stamp. The timed-timed case occurs when the addition of of strict-time information to the KPN/BKPN network connected to the T-SDF part has to be considered. We have four cases to consider:

(1) KPN/BKPN→T-SDF;

(2) T-KPN/T-BKPN→T-SDF;

(3) T-SDF→KPN/BKPN;

(4) T-SDF→T-KPN/T-BKPN.

For each case, we give the semantic of the converter channel for the writing and the reading sides. This includes the problem of time information injection for the timed-untimed connections. If the overlap of untimed/timed write and untimed/timed read semantics generate any inconsistency, the converter channel has to perform appropriate actions.

We start with the KPN/BKPN→T-SDF connection. From the writing side, the KPN/BKPN characteristics of the untimed network does not carry semantical problems. The converter channel can be either configured as an unbounded FIFO (as the HetSC `uc_inf_fifo` channel), such that the untimed part will never block. Or it can be configured as a bounded FIFO (as the HetSC `uc_fifo` channel), such that the untimed part can block. In any case, there is no loss of data. Syntactically, this is achieved by connecting the port `out` of a writing module `module` with the code line

```
module.out(sig.source_fifo (size));        (6)
```

to a converter channel `sig`. If the integer parameter `size` is omitted, the internal buffer of the converter channel will be unbounded.

The time-domain and communication-domain adaptations are closely related. The KPN/BKPN not only expects to do a nondestructive write, but also a destructive read which frees space in the internal buffer. Therefore, data consumption is needed on the reading side. This might appear like a contradiction, since, from the T-SDF side, the read is nonconsuming (as well as nonblocking). However, the read is nonconsuming only for the reads done at the same time stamp (t). From a sampling time stamp (t) to the next sampling time stamp ($t+T$), after a sampling period T, a data token is deallocated and frees space in the internal buffer, establishing a semantical coherence.

On the T-SDF reading side, a token is indeed consumed, but can be read as many times as wanted, during a sampling period T. That way, time information is injected in the untimed part. If, for instance, the converter channel has internal buffer size 2, and the sampling period is 1 millisecond, then the original untimed part will be able to initially write up to 2 tokens and compute until the point when it tries to write the third token. Then, it has to wait for 1 millisecond.

However, corner conditions can still lead to inconsistencies. In the KPN/BKPN→T-SDF case, the inconsistency comes when the internal buffer is empty and the read access of the T-SDF part has to consume a token. The converter channel offers several options for this case.

(i) *Error:* an error is raised and the simulation is stopped.

(ii) *Constant:* a prefixed value is returned. This value remains constant during the simulation. A default value is defined for each data type (i.e., `false` for `Bool` or `SC_LOGIC_X` for `sc_logic`), but can be overwritten by the designer.

(iii) *Hold:* the last value read is returned. It can be seen as a variation of the previous one, where the returned value, instead being constant, is the last token which could be consumed.

In the last two cases, warnings are raised. The designer can choose the desired behaviour by calling the method

```
converterchannel.setFIFO2SDFemptybuffer
(< option >);        (7)
```

where < `option` > stands for either `error`, `constant` or `hold`, where `error` is set by default. This ensures that the designer will be aware of this corner situation resulting from the specification and/or simulation.

The timed-timed T-KPN/T-BKPN→T-SDF connection can be treated basically like the KPN/BKPN→T-SDF case, including the handling of empty internal buffers. In particular, there is no difference regarding the syntax. There is, however, a subtle semantical difference regarding the empty FIFO corner case. In the untimed case, there are basically two causes for empty FIFOs:

(i) The writing process is finite, thus producing only a finite number of token. This will cause an empty FIFO if the simulation time is long enough. Depending on the application, choosing the consume & constant option here can make sense.

(ii) A (partial) deadlock in the KPN/BKPN side involving the process which writes the converter channel. This case is probably treated best with the default consume & error behaviour.

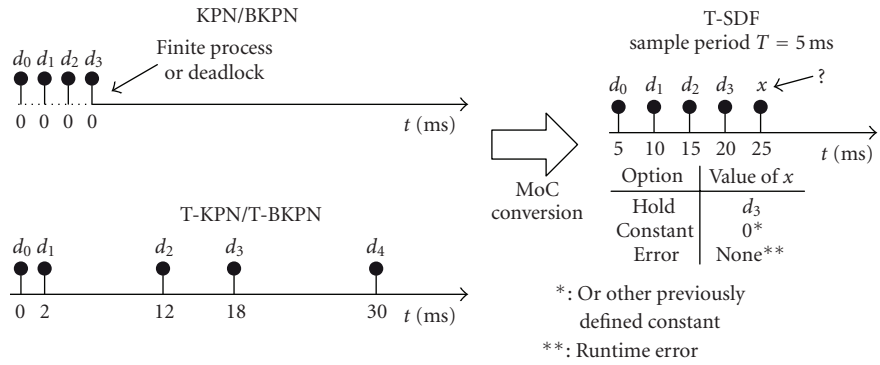


Figure 8: Time-domain behaviour for the KPN→T-SDF and BKPN→T-SDF conversion.

In the timed case, we have a *third cause*, namely the production rate of the writing side being insufficient compared to the consumption throughput of the T-SDF part. In fact, this is a case where the consume & hold behaviour makes perfect sense.

Note that reasons for the KPN/BKPN part to become timed may also be subtle. On one hand processes involved could contain explicit `wait()` statements. On the other hand, there is also the possibility of a timing injection from a third system part communicating with the, originally untimed, KPN/BKPN part. The variety of reasons for empty internal buffers makes it obvious that there is no a-priori intuitive solution to this problem, and the designer has to decide which of the three options offered by the converter channel suits best.

Figure 8 illustrates the conversion process described with an example showing the result depending on the chosen option. The untimed part, either KPN or BKPN, produces the sequence of tokens $\{d_0, d_1, d_2, d_3\}$. The order information, that is, the indexing of the values, is the only relevant time information. For instance, the specification of the untimed part only forces token d_1 to appear before token d_2, but forces nothing else about their SystemC time tags. In Figure 8, a possible time tag assignment consisting in all the tokens having the 0 millisecond time stamp is shown. Note that this is consistent since this order can be reflected by different deltas of the SystemC simulation. After producing d_3, the KPN/BKPN part stops producing tokens.

Assuming that the T-SDF part has a consumption rate 1 with a cluster time of $T = 5$ ms, the converter channel semantic consists in consuming each new token present in the inner buffer of the converter channel at a T pace. Since this pace of T-SDF cluster execution is unstoppable, in this example, a point in time is found (SystemC time tag of 25 seconds) when the T-SDF part needs to consume a data but the inner buffer is empty. In this situation the converter channel exhibits its flexibility. In order to still enable automatic conversion, the converter channel presents the default behaviour, that is, it raises an error. This could coincide with the designers intention, for instance, to detect a deadlock situation in the untimed part.

However, let us assume that the timed part models the sourcing of a kind of irregular pulse signal composed first by the finite subsequence $\{d_0, d_1, d_2, d_3\}$, followed by the infinite subsequence {'0', '0', '0', ...}. In this case, it makes sense to configure the converter channel to adopt the *constant* semantic for the corner case (X = '0'), instead of stopping the simulation and raising an error.

On the left bottom side of Figure 8, a similar case where the KPN/BKPN part suffered a time annotation, becoming a T-KPN/T-BKPN part, is shown. In this case, the empty buffer condition is not caused by a deadlock, but is the result of slow data token generation by the T-KPN/T-BKPN part. Token d_4 is generated at 30 seconds, too late to avoid the corner situation given at $t = 25$ seconds. In a case like this, the *hold* semantic for the corner case ($X = d_3$) is useful to model a sampling and hold behaviour.

Let us now turn to the timed-untimed T-SDF→KPN/BKPN conversion, where we have to introduce the FIFO semantic from the reading side. Here, the T-SDF cluster behaves as a kind of master which transfers tokens to the untimed part at the pace of the sampling time. We decided to use an unbounded internal buffer as the default case, but the designer can also limit its size by calling the method.

```
converterchannel.setReadingFIFOSize(int size).
```
(8)

The modification of the sense of data transfer changes the location of the semantic inconsistency. While the empty buffer does not represent a problem because the KPN/BKPN part remains blocked till the next data arrival, the full buffer condition is problematic, since the T-SDF has to write at its unstoppable pace. This inconsistency only appears when using a bounded internal buffer, and can appear due to several causes, similar to the ones presented for the KPN/BKPN→T-SDF case and the T-KPN/T-BKPN→T-SDF case. Namely, the KPN/BKPN part can be a finite process that finishes its execution after having consumed a finite number of tokens. It can also present a deadlock or partial deadlock preventing the consumption of tokens. For the case of a timed KPN/BKPN part, the consuming task can also present a throughput slower than the production throughput of the T-SDF part.

Therefore, in a T-SDF→(timed or untimed) KPN/BKPN connection, the write access semantic of the converter

channel allocates a new token in the internal buffer whenever there is room for it. If the internal buffer is full, the following options are available:

(i) *error:* an error is raised and the simulation is stopped;

(ii) *discardOldest:* the next token to be read by the consumer (at the "beginning" of the buffer) is removed, the buffer content is shifted forward and the current token (passed as parameter of the write access) is added to the "end" of the buffer;

(iii) *discardCurrent:* the current token is discarded and the internal buffer remains untouched.

The error option is again set by default. Figure 9 shows an example of a T-SDF→KPN/BKPN conversion, where the T-SDF writing side faces a full internal FIFO at $t = 25$ ms. This example also demonstrates, how the T-SDF side injects timing into the KPN/BKPN part: After unblocking at $t = 27$ ms, the KPN/BKPN part consumes all the token in the internal FIFO. After that, it is blocked until the next writing access of the T-SDF part.

6. APPLICATION EXAMPLE

In this section, we give an example on how converter channels can be used with respect to mixed level simulation and design space exploration. The example deals with the evaluation of a software-defined radio (SDR). An overview of the system is shown in Figure 10, and is basically a simplification of the example given in the introduction. The RF input signal is mixed with a sine wave which has the same frequency as the carrier signal and the result is processed by a lowpass filter. After that, the demodulation is done by software and an analogue demodulator, to compare the results. The input of the software demodulator is an integer with fixed bit width. Algorithm 2 shows the corresponding top-level SystemC code using two converter channels.

Regarding design space exploration and mixed level simulation, this example gives rise to the following tasks (the numbers refer to those in Figure 10):

(1) realising the lowpass filter either as a (behavioral) T-SDF-module or as an electrical network;

(2) realising the software demodulator either as a process network or as a DE module;

(3) varying the bit width of the input of the software demodulator;

(4) realising the analogue demodulator either as a (behavioral) T-SDF-module or as an electrical network.

To make the matter more complicated, any subset of these tasks can be done in parallel. It is obvious that the effort for manual inserting (and adapting) the appropriate converters would be significant. Let us assume an initial model with the lowpass filter and the analogue demodulator modelled as T-SDF modules and the software demodulator modelled within the BKPN MoC, taking `sc_int < bitwidth >` inputs. We then would need a T-SDF→BKPN converter from the output of the lowpass filter to the software modulator, which would also convert double values to `sc_int < bitwidth >` values. Now, executing the tasks above would cause the need for the following respective manual conversion activities.

(1) Realising the lowpass filter as an electrical network: Insert a T-SDF→CT-NET converter between the mixer and the lowpass filter. Replace the initial T-SDF→BKPN converter by a CT-NET→BKPN converter, which also converts `double` to `sc_int < bitwidth >`. Instantiate appropriate signals to connect them.

(2) Realising the software demodulator as a DE module: Replace the initial T-SDF→BKPN converter by a T-SDF→DE converter, which also converts `double` to `sc_int < bitwidth >`. Note that using a SystemC-AMS converter port makes no sense here, since the analogue demodulator is still in the T-SDF domain. Instantiate appropriate signals to connect them.

(3) Varying the bit width of the input of the software demodulator: Change the output of the T-SDF→BKPN converter and the type of the signal which connects it to the software demodulator to `sc_int < new_bitwidth >`. The data type conversion algorithm of the converter must also be altered slightly. Of course, using a parametrisable converter regarding the bit width would make things easier here. In that case this design space exploration task could be steered similar to Algorithm 2.

(4) Realising the analogue demodulator as an electrical network: Insert a T-SDF→CT-NET converter between the lowpass filter and the analogue demodulator. Instantiate an appropriate signal to connect them.

By using converter channels, however, the code for each of the possible variants would be very similar to Algorithm 2. The value of the variable `bitwidth` would change, and the class names of the respective modules (e.g., changing `lowpass_behavioural` to `lowpass_electrical`). This very simple example shows the convenience converter channels offer to the designer.

7. CONCLUSION AND FUTURE WORK

In this article, an overview on the ongoing work on converter channels has been given. Converter channels will provide the designer with a convenient tool to connect system parts using different MoCs and data types. They will quickly and safely solve the syntactical and semantical adaptations that are required by mixed-level simulation and design space exploration. Specifically, the syntactical solution of converter channels will save the manual refinement of those connections. In addition, converter channels provide suitable conversion semantics, which are not always straightforward, since they deal with adaptations in the time, communication, and data type domains. Thus, manual coding for adaptation

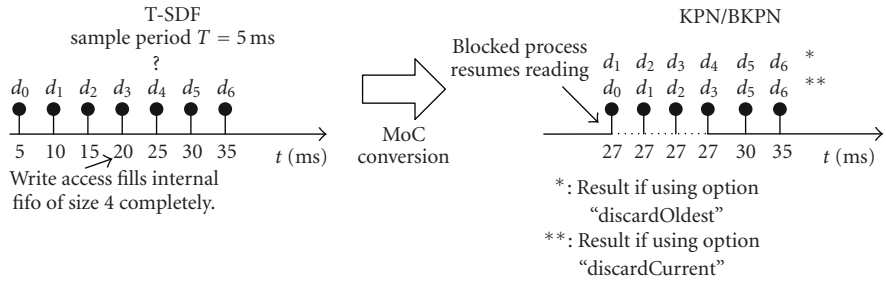


Figure 9: Time-domain behaviour for the T-SDF→KPN and T-SDF→BKPN conversion.

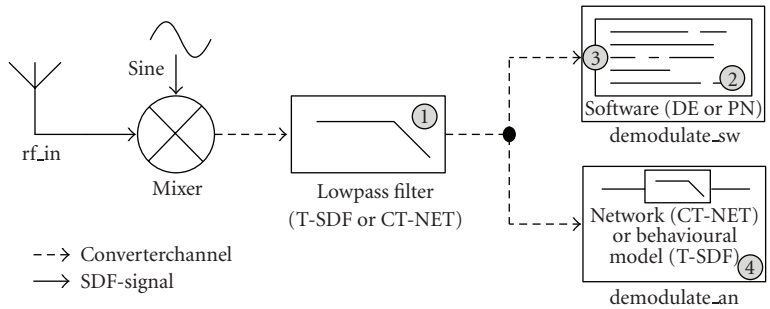


Figure 10: SDR application example.

```
bitwidth = 8;

sca_sdf_signal < double > rf_in; // incoming RF-signal
sca_sdf_signal < double > sine; // sine waverf_in
converterchannel < double > mixedsig;
  // RF-signal multiplied with sine-wave
converterchannel < double, sc_int < bitwidth >> lp_out;
  //output of lp-filter

lp_out.setRangeScaling (0., 1.);
  // assuming the value range within [−1, 1]

mixer mix( "mix"); // multiplies the two input signals
  mix.in1 (rf_in);
  mix.in2 (sine);
  mix.out (mixedsig);

lowpass_behavioural lp ("lp"); // lowpass filter, either
T-SDF-module
  lp.in (mixedsig); // or electrical network
  lp.out (lp_out);

demodulate_sw dem_sw ("dem_sw", bitwidth);
  rec_sw.in (lp_out);
  // software demodulator with sc_int < bitwidth > input

demodulate_an dem_an ("dem_an");
  rec_sw.in (lp_out);
  // analogue demodulator
```

Algorithm 2: SW-defined radio with converter channels.

would be quite time consuming for the system designer, who is usually more interested in the semantic of each system part, than in the glue semantic. The adaptations have been illustrated through several details on the usage of a converter channel and its internal structure, as well as, through a detailed explanation of the semantical issues of the (untimed and timed) KPN/BKPN↔T-SDF connection.

Future work will include finalising the support for further MoC connections, for example, the connection to linear electrical networks (CT-NET). Regarding the connection of CT-NET to T-SDF and DE modules, this is a straightforward task since SystemC-AMS provides many converter facilities for this. Another focus will be the use of converter channels to connect T-SDF models to TLM models.

Despite of ANDRES focussing on run-time reconfigurable systems, converter channels are not meant to adapt to changing communicating facilities during simulation time, for example, to react to a port which changes its bit width. This would go beyond the capabilities of SystemC itself in its current state. Nevertheless, this is an aspect which could be of interest in the future, since this also would allow for an automatic support of a dynamic brand of mixed-level simulation. In this case, the abstraction levels of certain system parts could even be changed during simulation time if the level of detail needed changes in certain situations.

ACKNOWLEDGMENT

This work is supported by the FP6-2005-IST-5 European project.

REFERENCES

[1] Open SystemC Initiative (OSCI), http://www.systemc.org.

[2] A. Vachoux, C. Grimm, and K. Einwich, "Towards analog and mixed-signal SOC design with systemC-AMS," in *Proceedings of the 2nd IEEE International Workshop on Electronic Design, Test and Applications (DELTA '04)*, pp. 97–102, IEEE Computer Society, Perth, Australia, January 2004.

[3] F. Herrera and E. Villar, "A framework for heterogeneous specification and design of electronic embedded systems in SystemC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, article 22, pp. 1–31, 2007.

[4] L. Cai and D. Gajski, "Transaction level modeling in system level design," Tech. Rep. 03-10, Center for Embedded Computer Systems, University of California, Irvine, Calif, USA, 2003.

[5] C. Grimm, "An introduction to modeling embedded analog/mixed-signal systems using SystemC ams extensions," June 2008.

[6] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[7] C. Grimm, R. Schroll, K. Waldschmidt, and F. Brame, "Mixed-level-simulation heterogener systeme," in *Proceedings of the ITG/GI/GMM-Workshop: Multi-Nature Systems*, Erfurt, Germany, February 2007.

[8] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, 1998.

[9] A. Jantsch, *Modeling Embedded Systems and SoC's*, Morgan Kaufmann, San Francisco, Calif, USA, 2004.

[10] I. Sander, "The ForSyDe standard library," Tech. Rep., Royal Technical School, KTH, Kista, Stockholm, April 2003.

[11] Accellera, "Verilog-AMS: Language Reference Manual," version 2.2, November 2004.

[12] P. Frey and D. O'Riordan, "Verilog-AMS: mixed-signal simulation and cross domain connect modules," in *Proceedings of the IEEE/ACM International Workshop on Behavioral Modeling and Simulation (BMAS '00)*, pp. 103–108, Orlando, Fla, USA, October 2000.

[13] IEEE, "IEEE Standard 1076.1: VHDL-AMS Language Reference Manual," 1999.

[14] L. Geppert, "Electronic design automation," *IEEE Spectrum*, vol. 37, no. 1, pp. 70–74, 2000.

[15] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy II)," Tech. Rep. UCB/EECS-2007-7, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Calif, USA, January 2007.

[16] T. Weilkiens, *Systems Engineering with SysML/UML*, Morgan Kaufmann, San Francisco, Calif, USA, 2008.

[17] OMG, OMG SysML Specification, March 2007.

[18] T. A. Jhonson, C. J. J. Paredis, J. M. Jobe, and R. Burkhart, "Modeling continuous system dynamics in SysML," in *Proceedings of the ASME International Mechanical Engineering Congress and Exposition (IMECE '07)*, Seattle, Wash, USA, November 2007.

[19] A. Davare, D. Densmore, T. Meyerowitz, et al., "A next-generation design framework for platform-based design," in *Proceedings of the Conference on Using Hardware Design and Verification Languages (DVCon '07)*, San Jose, Calif, USA, February 2007.

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

[21] H. D. Patel and S. K. Shukla, *SystemC Kernel Extensions for Heterogeneous System Modeling*, Springer, New York, NY, USA, 2004.

[22] H. D. Patel, D. Mathaikutty, and S. K. Shukla, "Implementing multi-moc extensions for SystemC: adding CSP and FSM kernels for heterogeneous modelling," Tech. Rep., FERMAT Research Laboratory, Virginia Tech, Blacksburg, Va, USA, June 2004.

[23] H. Al-Junaid and T. Kazmierski, "An Analogue and Mixed-Signal Extension to SystemC," http://eprints.ecs.soton.ac.uk/10644.

[24] S. Orcioni, G. Biagetti, and M. Conti, "SystemC-WMS: mixed signal simulation based on wave exchanges," in *Applications of Specification and Design Languages for SoCs*, pp. 171–185, Springer, New York, NY, USA, 2006.

[25] A. Herrholz, F. Oppenheimer, P. A. Hartmann, et al., "ANDRES-analysis and design of run-time reconfigurable, heterogeneous systems," in *Proceedings of the Design, Automation and Test in Europe (DATE '07)*, pp. 64–71, Nice, France, April 2007.

[26] A. Schallenberg, F. Oppenheimer, and W. Nebel, "Designing for dynamic partially reconfigurable FPGAs with SystemC and OSSS," in *Advances in Design and Specification Languages for SoCs: Part III*, The Chip Design Languages (ChDL) Series, pp. 183–198, Springer, Dordrecht, The Netherlands, 2005, C/C++-Based System Design.

[27] H. Posadas, F. Herrera, V. Fernández, P. Sánchez, E. Villar, and F. Blasco, "Single source design environment for embedded

systems based on SystemC," *Design Automation for Embedded Systems*, vol. 9, no. 4, pp. 293–312, 2004.

[28] N. Savoiu, S. K. Shukla, and R. K. Gupta, "Efficient simulation of synthesis-oriented system level designs," in *Proceedings of the 15th International Symposium on Systems Synthesis (ISSS '02)*, pp. 168–173, Kyoto, Japan, October 2002.

[29] F. Herrera, E. Villar, C. Grimm, M. Damm, and J. Haase, "A general approach to the interoperability of HetSC and SystemC-AMS," in *Proceedings of the Forum on Design Languages (FDL '07)*, Barcelona, Spain, September 2007.

[30] R. Schroll, *Design komplexer heterogener Systeme mit Polymorphen Signalen*, Ph.D. thesis, Institut für Informatik, Universität Frankfurt, Frankfurt, Germany, 2007.

[31] M. Damm, F. Herrera, J. Haase, E. Villar, and C. Grimm, "Using converter channels within a top-down design flow in SystemC," in *Proceedings of the 15th Austrian Workshop on Microelectronics (Austrochip '07)*, Graz, Austria, October 2007.

[32] F. Herrera, P. Sanchez, and E. Villar, "Heterogeneous system-level specification in SystemC," in *Advances in Design and Specification Languages for SoCs*, P. Boulet, Ed., CHDL Series, Springer, New York, NY, USA, 2005.

[33] F. Herrera and E. Villar, "Mixing synchronous reactive and untimed mocs in SystemC," in *Applications of Specification and Design Languages for SoCs*, A. Vachoux, Ed., CHDL Series, Springer, New York, NY, USA, 2006.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 712329, 18 pages
doi:10.1155/2008/712329

Research Article

Combining UML2 Application and SystemC Platform Modelling for Performance Evaluation of Real-Time Embedded Systems

Jari Kreku,[1] Mika Hoppari,[1] Tuomo Kestilä,[1] Yang Qu,[2] Juha-Pekka Soininen,[1] Per Andersson,[3] and Kari Tiensyrjä[1]

[1] *Communication Platforms, Technical Research Centre of Finland (VTT), Kaitoväylä 1, FI-90571 Oulu, Finland*
[2] *Nokia Device R&D, Elektroniikkatie 13, FI-90571 Oulu, Finland*
[3] *Department of Computer Science, Faculty of Engineering, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden*

Correspondence should be addressed to Jari Kreku, jari.kreku@vtt.fi

Received 1 October 2007; Revised 19 February 2008; Accepted 16 June 2008

Recommended by Eugenio Villar

Future mobile devices will be based on heterogeneous multiprocessing platforms accommodating several stand-alone applications. The network-on-chip communication and device networking combine the design challenges of conventional distributed systems and resource constrained real-time embedded systems. Interoperable design space exploration for both the application and platform development is required. Application designer needs abstract platform models to rapidly check the feasibility of a new feature or application. Platform designer needs abstract application models for defining platform computation and communication capacities. We propose a layered UML application/workload and SystemC platform modelling approach that allow application and platform to be modelled at several levels of abstraction, which enables early performance evaluation of the resulting system. The overall approach has been experimented with a mobile video player case study, while different load extraction methods have been validated by applying them to MPEG-4 encoder, Quake2 3D game, and MP3 decoder case studies previously.

1. INTRODUCTION

Future handheld mobile multimedia terminals will merge features of several currently independent mobile devices, for example, phone, music player, television, movie player, desktop, and Internet tablet. They accommodate a large number of on-terminal and/or downloadable applications that offer the user with services, whose contents may be provided by servers anywhere on Earth. The sets and types of applications running on the terminal are dependent on the context of the user. To deliver requested services to the user, some of the applications run sequentially and independently, while many others execute concurrently and interact with each others.

The digital processing architectures of terminals will evolve from current system-on-chips (SoCs) and multi-processor-SoCs with a few processor cores to massively parallel computers that consist mostly of heterogeneous subsystems, but may also contain homogeneous computing subsystems. The network-on-chip (NoC) communication paradigm will replace the bus-based communication to allow scalability, but it will increase uncertainties due to latencies in case large centralized or external memories are required.

The application and architecture development trends will increase the overall complexity of system development by orders of magnitude. The optimisation of performance, energy, and cost of the battery-powered devices while respecting the user expectations is of vital importance on one hand. On the other, the costs, risks, and time of system development require flexibility, which will be achieved through programmable/adaptable computing resources, configurable memory and communication architecture, and design methodology approach; and tools that span from application use cases until implementation design. Design methodology approaches for mobile devices have evolved from the application-specific integrated circuit (ASIC) style in 80's to platform-based design in late 90's and model-based design was introduced recently.

In the ASIC style, designers take a (architectural) specification, create a microarchitecture description, and

synthesise/optimise it for speed (clock frequency), area (gate count), and power (e.g., modes and clock gating).

Platform-based design addresses the challenges of increasing design complexity of SoCs that consist typically of a few processor cores, hardware accelerators, memories, and I/O peripherals communicating through a shared bus. While the emphasis is on intellectual property (IP) design and integration, the function-architecture codesign and (micro-) architecture exploration already pave the way to the model-based approach that is the research direction today. To alleviate the scalability problems of the shared bus, the NoC architecture paradigm proposes communication centric approach for systems requiring multiple processors or integration of multiple SoCs.

Model-based approaches extend the separation of the application and execution platform modelling further. Usually the specify-explore-refine paradigm following the principles of the Y-chart model [1] is applied. In other words, a model of application is mapped onto a model of platform and the resulting allocated model is analysed. The computation and communication modelling are separated on both the application and platform sides. Recent trend is service-orientation, where the end user interactions and the associated applications are modelled in terms of services required from the underlying execution platform [2]. An obvious consequence is that the execution platform also needs to be modelled in terms of services it provides for the applications.

Both the application and platform designers are facing an abundant number of design alternatives and need systematic approaches for the exploration of the design space. For example, an application designer has to know early whether a new application or a feature is feasible on the target platform. A platform designer must be able to analyse the impacts of next generation applications on the platform even before the applications are implemented. Efficient methods and tools for early system-level performance analysis are necessary to avoid wrong decisions at the critical stage of system development.

The performance analysis models are required to capture both the characteristics of the application functionality and the architectural resources needed for the execution. Using models at a too low level of abstraction, for example, register-transfer-level (RTL) or instruction-set simulation (ISS) is not feasible: although giving accurate results, due to the vast amount of details needed the modelling effort is heavy and simulation times are long. Some high-level abstraction approaches like queuing networks (QNs) and its variants fail to exhibit the characteristics of the execution platforms.

In this paper, we present VTT_ABSOLUT2.2% (VTT ABstract inStruction wOrkLoad & execUtion plaTform UML2/SystemC2-based performance simulation)—it is a model-based approach for system-level design that is capable of performance evaluation of future real-time embedded systems and provides early information for development decisions. Applications are modelled in either unified modelling language (UML) or SystemC [3] domain as workloads consisting of load primitives. Platform models are cycle-approximate transaction-level SystemC models. Mapping between UML application models and the SystemC platform models is based on automatic generation of simulation models for system-level performance evaluation. The workload models reflect accurately the control structures of the applications, but the computing and communication loads are abstractions derived either analytically from measured traces or using a source code-compilation approach called VTT_ABSINTH (VTT ABStract INstruction exTraction Helper) (Figure 1). The execution platform model is configured from a library of performance models using VTT_COGNAC (VTT COnfiguration GeNerator for Absolut performanCe simulation). The executable simulation model is based on the open source open SystemC initiative (OSCI) SystemC library, extended with configurable instrumentation and called VTT_BEER (VTT Binary pErformance EvaluatoR). Finally, the simulation results can be selected for analysis and viewed using VTT_VODKA (VTT Viewer of collecteD Key information for Analysis).

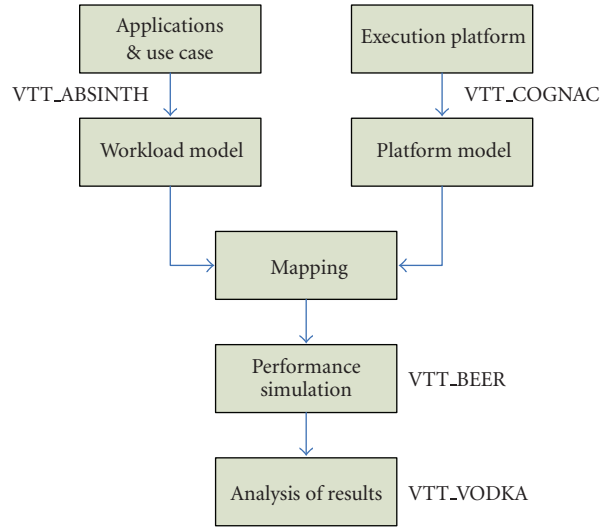


Figure 1: Y-chart model of plain VTT_ABSOLUT.

The tool support is based on a commercial UML2 tool, Telelogic Tau G2, and an open source SystemC simulation tool of OSCI.

The approach enables early performance evaluation, exhibits light modelling effort, allows fast exploration iteration, and reuses application and platform models. It also provides performance results that are accurate enough for system-level exploration.

2. RELATED WORK

Performance evaluation has been approached in many ways at different levels of refinement. Some other research approaches aiming at similar goals to ours by different modelling and simulation approaches are described briefly in the sequel.

SPADE [4] implements a trace-driven, system-level cosimulation of application and architecture. The application is described by Kahn process networks using YAPI [5].

Symbolic instruction traces generated by the application are interpreted by architecture models to reveal timing behaviour. Abstract, instruction-accurate performance models are used for describing architectures.

The Artemis work [6] extends the work described in [4] by introducing the concept of virtual processors and bounded buffers. One drawback of restricting the designer to using Kahn process networks is the inability to model time-dependent behaviour. In the developed Sesame modelling methodology, a designer first selects candidate architectures using analytical modelling and multiobjective optimization. The system-level simulation environment allows for architectural exploration at different levels of abstraction. The high-level and architecture-independent application specifications are maintained by applying dataflow graphs in its intermediate mapping layer. These dataflow graphs take care of the runtime transformation of coarse-grained application-level events into finer grained architecture-level events that drive the architecture model components.

The basic principle of the TAPES [7] performance evaluation approach is to abstract the involved functionalities by processing latencies and to cover only the interaction of the associated subfunctions on the architecture. These interactions are represented as inter-SoC-module transactions, without actually running the corresponding program code. This abstraction enables higher simulation speed than an annotated, fully-fledged functional model. Each subfunction is captured as a sequence of transactions, also referred to as trace. The binding decision for the subfunctions is considered by storing the corresponding trace in the respective architectural resource. A resource may contain several traces, one per each subfunction that is bound to it. The application is then simulated by forwarding packet references through the system and triggering the traces that are required for processing particular data packets in the respective SoC modules.

MESH [8] looks at resources (hardware building blocks), software, and schedulers/protocols as three abstraction levels that are modelled by software threads on the evaluation host. Hardware is represented by continuously activated, rate-based threads, whereas threads for software and schedulers have no guaranteed activation patterns. The software threads contain annotations describing the hardware requirements, so-called time budgets that are arbitrated by scheduler threads. Software time budgets are derived beforehand by estimation or profiling. The resolution of a time budget is a design parameter and can vary from single compute cycles to task-level periods. The advance of simulation time is driven by the periodic hardware threads. The scheduler threads synchronize the available time budgets with the requirements of the software threads.

SpecC [9] defines a methodology for system design including architecture exploration, communication synthesis, validation, and implementation. SpecC can be considered as a specification and modelling language that has a rich support for many system design phases. Similar properties can be found also in SystemC language that is more widely adopted in high-level system modelling. Especially, the transaction-level modelling using SystemC has been adopted for performance modelling and simulation [10], and OSCI is finalising the version 2.0 of its SystemC Transaction-level Modelling Standard [11].

Posadas et al. in [12] present a POSIX-based SystemC RTOS library for timing estimation at system level. The library is based on a general and systematic methodology that takes as input the original SystemC source code without any modification and provides the estimation parameters by simply including the library within a usual simulation. As a consequence, the same models of computation used during system design are preserved and all simulation conditions are maintained. The method exploits the advantages of dynamic analysis: easy management of unpredictable data-dependent conditions, and computational efficiency compared with other alternatives (ISS or RTL simulation, without the need for software (SW) generation and compilation and hardware (HW) synthesis).

Koski [13] is a UML-based automated SoC design methodology focusing on abstract modelling of application and architecture for early architecture exploration, methods to generate the models from the original design entry, system-level architecture exploration performing automatically allocation and mapping, tool chain supporting the defined methodology utilizing a graphical user interface, well-defined tool interfaces, a common intermediate format, and a simulation tool that combines abstract application and architecture models for cosimulation.

Ptolemy II [14] is a Java-based software framework developed as part of the Ptolemy project [15], supporting heterogeneous, concurrent modelling and design. Metropolis is a framework for platform-based design, which consists of an internal representation mechanism, the design methodology, and base tools for simulation and design imports [16].

MARTE [17] is a UML profile for model-driven development of real-time and embedded systems, which provides support for specification, design, and verification/validation. It is intended to replace the earlier UML schedulability, performance, and time (SPT) profile.

Our approach differs from the SPADE and Artemis as to the way the application is modelled and abstracted. The UML-based workload model mimics truly the control structures of the applications, but the leaf level load data is presented like traces. Also the execution platform is modelled rather at transaction than instruction level. The TAPES approach uses transaction-level SystemC simulation like ours, but describes the application functionality as traces that are stored in the architectural resources. The layering approach of the MESH is somewhat similar to ours, but the way of obtaining the timing information differs; in MESH it is obtained by profiling on a host, while the simulation gives the timing information in our case. The SpecC approach estimates timing information for the simulation, too. The Koski approach uses libraries and compilation/synthesis on field programmable logic arrays (FPGAs) to extract timing, although it allows setting timing requirements during application modelling. The UML part of our approach could have been based on the MARTE profile; however, the profile was not publicly available at the time of this work.

3. UML2-SYSTEMC-BASED PERFORMANCE MODELLING

3.1. Overview

The performance modelling and evaluation approach of VTT_ABSOLUT2.2% follows the Y-chart model as depicted in Figure 1. The basic principle of our performance evaluation approach is as follows [18]. The layered hierarchical workload models represent the computation and communication loads the applications cause on the platform when executed. The layered hierarchical platform models represent the computation and communication capacities the platform offers to the applications. The workload models are mapped onto the platform models and the resulting system model is simulated at transaction-level to obtain performance data.

The starting points for the performance modelling are the end-user requirements of the system. These are modelled as a service-oriented application model, which has a layered hierarchy. The top layer consists of system level services visible to the user that are composed of subservices and divided further to primitive services. The functional simulation of the model in Telelogic Tau UML2 tool provides sequence diagrams which are needed for verification and for building the workload model.

The purpose of workload modelling is to illustrate the load an application causes to an execution platform when executed. Workload models are nonfunctional in the sense that they do not perform the calculations or operations of the original application. Workload modelling enables performance evaluation already in the early phases of the design process, because the models do not require that the applications are finalized. Workload modelling also enhances simulation speed as the functionality is not simulated and models can typically be easily modified to quickly evaluate various use cases.

Platform modelling comprises the description of both hardware and platform software (middleware) components and interconnections that are needed for performance simulation. Like workload modelling, platform modelling considers hierarchical and repetitive structures to exploit topology and parallelism. The resulting models provide interfaces, through which the workload models use the resources and services provided by the platform [19].

The workload models are created with UML or SystemC, while the platform models are based on SystemC only. If the workload modelling is done with a UML tool, the models have to be transformed into SystemC. UML is a standard language used in software development and thus the possibility to use UML-based workload models enables reuse of existing UML application models. This ultimately reduces the effort required for workload modelling and makes the performance simulation approach more accessible in general. The use of UML is also beneficial because the models are visual and easier to understand for others besides the person who created them.

Figure 2 depicts the flow from UML2 application models to generated SystemC workload models [20]. The entire hierarchy of workload models—applications, processes, functions, and so forth—are collected in a class or package diagram, which presents the associations, dependences, and compositions of the workloads. Control inside the application, process and function workloads is described with state machine diagrams. Composite structure diagrams are used to connect the control implementation with the corresponding workload model. Section 6 presents examples of how the composite structure and state machine diagrams are used in our approach for implementing the workload models. All workload model layers, with the possible exception of the load primitive layer, are implemented in the UML model.

A skeleton model of the platform is manually created in the UML model. This facilitates mapping between the workload models with service requirements and the platform models with service provisions. The skeleton model describes the components and services available in the platform and thus enables the use of these services from the workloads. In the mapping phase, each workload entity is linked to a processor or other component, which is able to provide the services required by that entity. This is realized in the UML model using composite structure diagrams, for example.

Transformation to SystemC is triggered from the Telelogic Tau tool and it is based on the approach developed by Lund University, which is described in [21]. The generator produces SystemC code files, which include SystemC modules of classes and channels required for communication, and Makefiles for building the models. However, we build the SystemC workloads together with the platform model and thus do not use the generated Makefiles. The load primitive layer, if not modelled in the UML domain, is implemented in the SystemC workload refinement phase. This includes either writing read(), write(), and execute() service calls manually into the models or using a suitable automatic tool for the job.

The form of the UML workload models is quite flexible and only the following criteria have to be met:

(i) the platform model provides its services via functions and the SystemC workloads generated from the UML workload models must be able to use them. This can be achieved via signals in state machine diagrams, for example,

(ii) only diagrams supported by Lund University 's code generator can be used [21].

After mapping the workloads to the platform, the models can be combined for transaction-level performance simulation in SystemC. Based on the simulation results, we can analyse, for example, processor utilisation, bus or memory traffic, and execution time.

3.2. Workload model

Workload models are used for characterising the control flow and the loads of the data processing and communication of applications on the execution platform. Therefore the models can be created and simulated before the applications are finalized, enabling early performance evaluation. As opposed to most of the performance simulation approaches, the workload models do not contain timing information. It

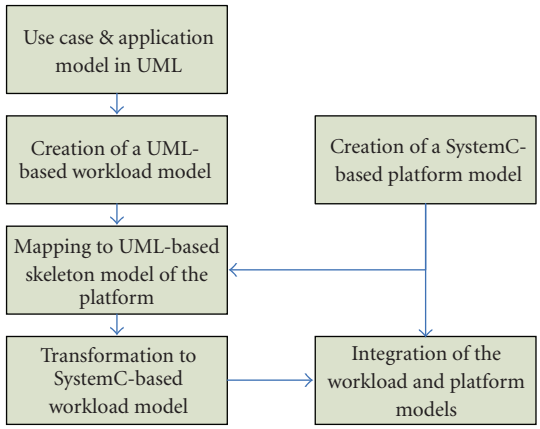


Figure 2: Transformation from UML2 application model to SystemC workload model.

is left to the platform model to find out how long it takes to process the workloads. This arrangement results in enhanced modelling and simulation speed. It is also easy to modify the models, which facilitates easier evaluation of various use cases with minor differences. For example, it is possible to parameterise the models so that the execution order of applications varies from one use case to another.

The workload models have a hierarchical structure, where top-level workload model W divides into application workloads A_i, $1 \le i \le n$ for different processing units of the physical architecture model (Figure 3):

$$W = \{C_a, A_1, A_2, \ldots, A_n\}, \tag{1}$$

where C_a denotes the common control between the workloads, which takes care of the concurrent execution of loads mapped to different processors. n is the number of application workloads under the top-level workload.

Each application workload A_i is constructed of one or more processes P_i:

$$A_i = \{C_p, P_1, P_2, \ldots, P_n\}, \tag{2}$$

where C_p corresponds to the control between the processes.

The structure of the main workload model and the application workloads is depicted in the UML diagram of Figure 4. The application and process control are shown as classes in the diagram; however, they may be implemented using, for example, standard C++ control structures in SystemC-based workload models.

The processes are comprised of function workloads i:

$$P_i = \{C_f, F_1, F_2, \ldots, F_n\}, \tag{3}$$

where C_f is control and describes the relations of the functions, for example, branches and loops. The operating system models of the platform handle workload scheduling at the process level.

Function workload models are basically control flow graphs

$$F_i = (V, G), \tag{4}$$

where nodes $v_i \in V$ are basic blocks and arcs $g_i \in G$ are branches. Basic blocks are ordered sets of load primitives used for load characterization. Load primitives are abstract instructions *read* and *write* for modelling memory accesses and *execute* for modelling data processing.

Process and function workload models can also be statistical. In this case the model will describe the total number of different types of load primitives and the control is a statistical distribution for the primitives (Figure 5). This is beneficial in case the chosen load extraction method is not accurate enough so that functions and/or basic blocks could be modelled in detail. Less important, for example, background, workloads can also be modelled this way for reducing the modelling effort. Workload models using deterministic process models but statistical function models are more accurate than those using statistical process models. Models, which are deterministic down to basic block level, are of course the most accurate.

3.3. Execution platform model

The platform model is an abstracted hierarchical representation of the actual platform architecture. It contains cycle-approximate timing information along with structural and behavioural aspects. The platform model is composed of three layers: component layer, subsystem layer, and platform architecture layer (Figure 6). Each layer has its own services, which are abstraction views of the architecture models. They describe the platform behaviours and related attributes, for example, performance, but hide other details. Services in the subsystem and platform architecture layers can be invoked by workload models. High-level services are built on low-level services, and they can also use the services at the same level. Each service might have many different implementations. This makes the design space exploration process easier, because replacing components or platforms by others could be easily done as long as they implement the same services.

3.3.1. Component layer

This layer consists of processing (e.g., processors, DSPs, dedicated hardware, and reconfigurable logic), storage, and interconnection (e.g., bus and network structure) elements. An element must implement one or more types of component-layer services. For example, a network interface component should implement both master and slave services. In addition, some elements need to implement services that are not explicitly defined in component-layer services, for instance, a bus will support arbitration and a network will support routing.

The component-layer read, write, and execute services are the primitive services, based on which higher-level services are built. The processing elements in the component layer realize the low-level workload-platform interface, through which the load primitives are transferred from the workload side. The processing element models will then generate accesses to the interconnections and slaves as appropriate.

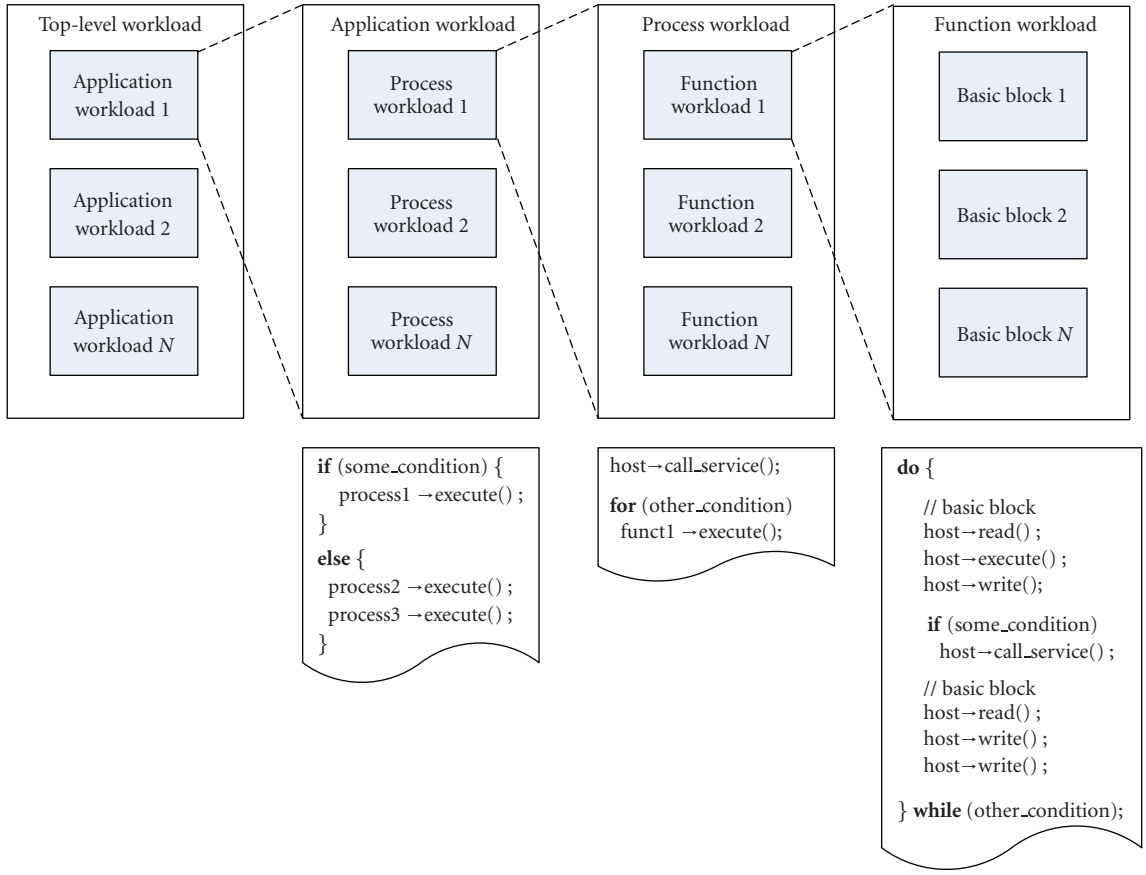


Figure 3: Workload models have a hierarchical structure.

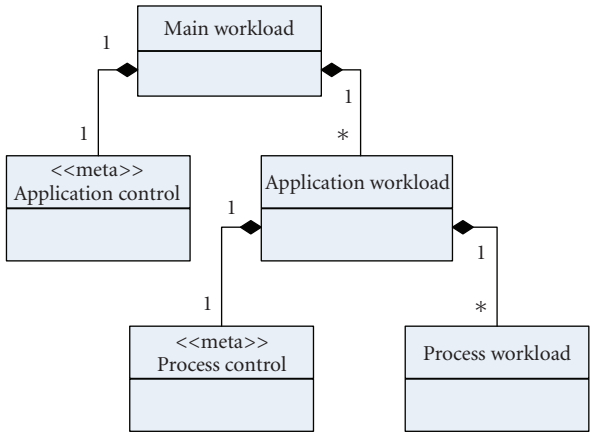


Figure 4: The top-level and application workloads consist of one or more lower level items and control.

All the component models contain cycle-approximate or cycle-accurate timing information. Specifically, the data path of processing units is not modelled in detail; instead the processor models have a cycles-per-instruction (CPI) value, which is used in estimating the execution time of the workloads. For example, the execution time for data processing instructions is the number of instructions to execute times CPI (Figure 7). Furthermore, caches and SDRAM page misses, for example, are modelled statistically since the workload models typically do not include accurate address information.

3.3.2. Subsystem layer

The subsystem layer is built on top of the component layer and describes the components of the system and how they are connected. The services used at this layer could include, for example, video preprocessing, decoding, and postprocessing for a video acceleration subsystem.

The model can be presented as a composition of structure diagrams that instantiates the elements taken from the library. The load of the application is executed on processing elements. The communication network connects the processing elements with each other. The processing elements are connected to the communication network via interfaces.

3.3.3. Platform architecture layer

The platform architecture layer is built on top of the subsystem layer by incorporating platform software and serves as the portals that link the workload models and the platforms in the mapping process. Platform-layer services

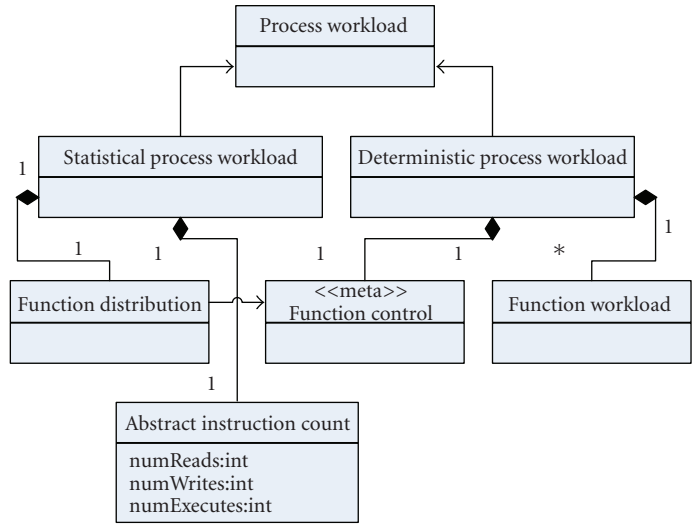


Figure 5: The process workloads can be either statistical or deterministic.

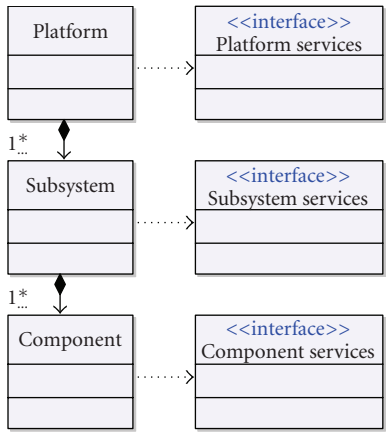


Figure 6: The execution platform model consists of platform, subsystem, and component layers.

consist of service declaration and instantiation information. The service declaration describes the functionalities that the platform can provide. Because a platform can provide the same service with quite different manners, the instantiation information describes how a service is instantiated in a platform.

The platform-layer services are divided into several categories with each category matching one application domain, for example, video processing, audio processing, and encryption/decryption. The OS system call services are in an individual domain, and as mentioned earlier they can also be invoked by other services at the same level. A number of platform-layer services are defined for each domain, and more could be added if necessary.

Application workloads typically call platform or subsystem level services, process workloads call subsystem services, and function workloads call component-level services. Ideally, all services required by the application are provided by the execution platform and there is a 1 : 1 mapping between the requirements and provisions. However, often this is not the case and the workloads need to use several lower-level services in combination to produce the desired effect.

Table 1: The low-level interface consists of functions intended for transferring load primitives between workload and platform models.

Interface function	Description
Read(A, W, B)	Read W words of B bits from address A
Write(A, W, B)	Write W words of B bits to address A
Execute(N)	Simulate N data processing instructions

3.4. Interface between workload and platform models

The platform model provides two interfaces for utilising its resources from the workload models. The low-level interface is intended for transferring load primitives and is depicted in Table 1. The functions of the low-level interface are blocking—in other words a load primitive level workload model is not able to issue further primitives before the previous primitives have been executed.

The high-level interface enables workload models to request services from the platform model (Table 2). These functions can be called from workload models between the function and application layers. The use_service() call is used to request the given service and it is nonblocking so that the workload model can continue while the service is being processed. Use_service() returns a unique service identifier, which can be given as a parameter to the blocking wait_service() call to wait until the requested service has been completed, if necessary.

In simple cases, the execution of workloads can be scheduled manually by hard-coding it to the models. However, typically the platform model includes one or more

```
void Gpp: :execute(unsigned count, const Process* owner)
{
 m_timer_exec→start();

 DEBUG_N"executing" <<count<<" instruction((s)");

 // calculate the number of cycles it takes to execute the instructions

 double cycles = count* m_props.cpi();
 unsigned u_cycles = (unsigned)std: :ceil(cycles + rounding_error);

 rounding_error+ = (cycles-u_cycles);

 DEBUG_N("cycles = " << u_cycles<<"; error = " << rounding_error);

 i_fetch(count);

 m_status→set_busy();
 wait_clk(u_cycles);
 m_status→set_idle();

 m_timer_exec→set(false, count);
}
```

Figure 7: Code extract showing how processor models calculate the execution time for data processing instructions.

Table 2: The services of the platform model are exploited via the high-level interface.

Interface function	Return value	Description
Use_service(*name*, *attr*)	Service identifier *id*	Request service *name* using *attr* as parameters
Wait_service(*id*)	N/A	Wait until the completion of service *id*

operating system (OS) models, which control access to the processing unit models of the platform by scheduling the execution of process workload models (Figure 8). The OS model provides both low-level and high-level interfaces to the workloads and relays interface function calls to the processor or other models which realize those interfaces. The OS model will allow only these process workloads which have been scheduled for execution to call the interface functions. Rescheduling of process workloads is performed periodically according to the scheduling policy implemented in the model.

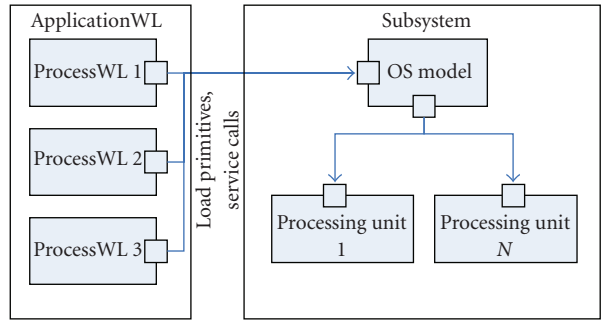


Figure 8: During simulation process workloads send load primitives and service calls to the platform model.

3.5. Transformation from UML to SystemC

Though it is possible to define a set of mapping rules from a pure UML model directly into SystemC code, it would give the engineer little influence on the mapping and most likely a less satisfactory result. Instead, we divide the mapping into three steps. All models are available and editable. This makes it possible for the engineer to have full control over the relevant details for the system under development and have the tool to manage all remaining details.

Step 1, vertical refinement transformation: In this step an initial UML description is refined to a UML description, which follows a UML profile for SystemC. This step will partly be carried out manually. To minimise the design effort, it is not required to tag the whole model. This means that a set of default values for the SystemC specific attributes of the UML profile have been defined. The default values provide a satisfactory mapping in the following transformation steps.

Step 2, vertical refinement transformation: in this step the model is transformed into a new UML description that only includes UML constructs with direct representations in SystemC, that is, classes, attributes, inheritance, and so on. Other constructs such as state machines are translated to the target language. During this step each state machine

is transformed to a class with methods that implement the behaviour of the states and transitions. In the first version of the tool, the resulting model is an untimed functional model. The mapping rules for UML to SystemC are beyond the scope of this paper and interested readers are referred to [21, 22].

In addition to removing UML only concepts, all relations in the model are made explicit. When a class is made active in UML it implies that the class will have its own thread of execution. In SystemC this is realized using SC_THREAD or SC_METHOD which implies that the class is an instance of the SystemC class sc_module. For example, a generalisation relation to the SystemC class sc_module is added from all active UML classes.

Step 3, horizontal transformation: in this step the UML model resulting from step 2 is transformed into a corresponding SystemC code. This transformation is a one–to-one correspondence between the UML model and the resulting SystemC code. This step is implemented using the existing C++ code generator from Telelogic and thus reuses its support for scope rules, header-file inclusion and make file generation without any modifications. If the generated code is to be read by humans, it is desirable to use the common SystemC macros when applicable. This requires a slight customisation of the syntax of the generated code. This is done using an agent, a mechanism which makes it possible for third-party executables to interact with the C++ Code generator in Telelogic Tau G2. The agent generates SystemC like module declarations, instead of a C++ class declarations, SC_MODULE(MyModule){$\cdots$} instead of class MyModule:public sc_module{$\cdots$}.

4. LOAD EXTRACTION

The workload models capture the control behaviour of the applications in the hierarchically layered structures. On the other hand, they abstract the details of data processing and communication as loads. To obtain the load information, three different techniques are currently used: analytical, measurement-based and source code-based. These can be used separately or in combination depending on what kind of descriptions of application algorithms are available.

The selection of which extraction method to use depends on (i) what kind of information is available of the application(s), (ii) how accurate the resulting models need be, and (iii) how much effort one is willing to use. In general, you will want to use as accurate method as possible for best results. On the other hand, if you have only limited information available of the applications, you may be forced to use the analytical method. If you are modelling small background load with a minor impact on the overall performance, it makes sense to use one of the faster extraction methods instead of the source code-based approach.

4.1. Analytical load modelling

Analytical load modelling method is currently the simplest way to create workload models but also probably the least accurate. As the name suggests it is based on analysis of the number of operations from an algorithm description or other suitable source.

Analytical modelling [23] consists of three phases. First, the number of data processing instructions and amount of memory traffic are analysed from the algorithm description. In principle, after this point we could write a simple three line workload model that would consist of one read(), one write(), and one execute() call. However, the ordering and block size of the operations usually affects the performance notably. Therefore in the second phase the total operation numbers are divided into smaller blocks. In the best case, this can also be done based on the algorithm description. In the worst case we can only create a number of uniformly distributed blocks of read(); execute(); write().

Let us consider the case of an MPEG4 video colour conversion algorithm as an example. Once the video resolution and the amount of bits per pixel for input and output are known it is possible to calculate the number of memory reads and writes per frame. For VGA resolution, YUV422 input and YUV420 output we get at least $640 \times 480 \times 16$ bits of data reads and $640 \times 480 \times 12$ bits of writes. It is also possible to estimate the efficiency of the implementation with a simple factor $n \leq 1$. The word length W (in bits) and burst length N (in words) of the target processor should also be taken into account so that the number of reads R and writes W for F frames in this case are

$$\begin{aligned} R &= \left\lceil \frac{640 \cdot 480 \cdot 16 \cdot F}{nWN} \right\rceil \\ W &= \left\lceil \frac{640 \cdot 480 \cdot 12 \cdot F}{nWN} \right\rceil, \end{aligned} \tag{5}$$

respectively. Since the operations performed for each pixel are known it is straightforward to estimate also the number of data processing instructions E. In this case the algorithm calculates an average of each two chrominance data points, so we have $640 \times 480/2$ calculations per frame. One addition and one division or shift by two, or two instructions in total, are required per each average, resulting to the following formula:

$$E = \left\lceil \frac{640 \cdot 480 \cdot F}{n} \right\rceil. \tag{6}$$

In this case, the algorithm must be executed so that data is available for at least two pixels at any point. Thus we can divide the model in the second phase so that firstly the data for two pixels is read, then the two pixels are processed and, finally, the new data is written out. This sequence is repeated until all pixels of all frames are processed. In pseudocode, this results to workload model shown in Algorithm 1.

However, it should be noted that this is not the only feasible way to divide the algorithm into smaller blocks.

The analytical method typically produces the most compact workload models with the least effort. However, the quality of the workload models depends a great deal on the use case (algorithm) being modelled. So far, the analytical approach for load extraction has been partially used in the modelling of Quake 2 3D game [23] and MPEG4 video encoding and decoding on the OMAP2 platform.

```
for(loop through all frames) {
    for(loop until all pixels of a
        single frame are processed) {
      read(two_pixels);
      execute(two_pixels);
      write(two_pixels);
    }
}
```

Algorithm 1

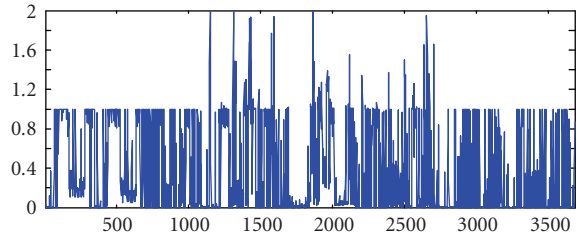


Figure 9: An example of a concatenated and merged utilisation curve.

4.2. Measurement-based load generation

The second method for creating the load models is to extract data from partial measurements or traces made of the use case [24]. This method can easily be automated, which allows rapid modelling of complex use cases with minimal manual work. It is also useful for performance analysis if the entire use case cannot be executed and monitored for some reason.

A method has been developed for generation of complex workload models from processor utilisation data. The complex use case means in this context a workload that cannot be measured with available system and that consists of several programs that can be measured or estimated alone. The first step of our approach is to measure processor utilisation data of these individual programs. The second step is to merge and concatenate the data according to the state sequence model of a complex use case. An example of this merging is shown in Figure 9. The merged utilisation curve is likely to have data points where the utilisation is more than 100%. This is not a problem because the value will only be used as a basis for estimating the number of load primitives for the workload model. The third step is to generate the workload models of the primitive samples of monitoring tool.

The workload models consist of read, write, and execute operations, whose amounts can be estimated for each sample point of the input utilisation data in the following way: let $X(n)$ be the combined trace data, which was composed by merging the measurement data, and N the number of samples in the combined trace. Furthermore, let t_{sp} be the sampling interval, C the number of cycles per second (i.e., processor clock frequency), c_{sp} the number of cycles per sample point, $C_{\mathrm{op}}^{R,W,E}$ the number of cycles per read, write, or execute operation, $\rho^{R,W,E}$ the percentages of read, write, and execute operations, and $Y^{R,W,E}(n)$ the amount of read, write, or execute operations per sample point.

Now, the time, t^R, spent for read operations during one sample point can be calculated as

$$t^R = t_{\mathrm{sp}}\rho^R. \tag{7}$$

During this time, the processor clock counts $t^R C$ cycles. This amount of cycles is sufficient to perform exactly $(t^R C)/c_{\mathrm{op}}^R$ read operations. When the combined trace data gives a utilisation level of X for the given sample point, n, we can assume that the number of read operations, $Y^R(n)$, for that sample point is $(t^R C)/c_{\mathrm{op}}^R X$, which can be rearranged as

$$Y^R(n) = C\frac{\rho^R}{c_{\mathrm{op}}^R}t_{\mathrm{sp}}X. \tag{8}$$

In general terms, this can be expressed as

$$\overline{Y}(n) = \begin{bmatrix} Y^R(n) \\ Y^W(n) \\ Y^E(n) \end{bmatrix} = C\left\lfloor \overline{B}t_{\mathrm{sp}}X(n)\right\rfloor, \quad n = 1,2,3,\ldots,N$$
$$\overline{B} = \begin{bmatrix} \dfrac{\rho^R}{c_{\mathrm{op}}^R} \\ \dfrac{\rho^W}{c_{\mathrm{op}}^W} \\ \dfrac{\rho^E}{c_{\mathrm{op}}^E} \end{bmatrix}. \tag{9}$$

In a shorter form this can also be expressed as

$$Y^{R,W,E}(n) = \left\lfloor \frac{c_{\mathrm{sp}}}{c_{\mathrm{op}}^{R,W,E}}\rho^{R,W,E}X(n)\right\rfloor \mid \; c_{\mathrm{sp}} = Ct_{\mathrm{sp}}\,, \quad n = 1,2,3,\ldots,N. \tag{10}$$

The execution of these operations is then spread uniformly across the given sample point and, finally, workload models containing the information for each sample point are composed.

The measurement-based approach has been successfully applied to complex use cases consisting of GPRS connection, Bluetooth download, SMS/MMS messaging, and video capturing among others [24]. For comparison purposes the same applications were executed in a mobile phone prototype based on the same platform. The performance of the platform was monitored and the average difference between measured and simulated processor utilisation was about 19%.

4.3. Source code-based load generation

Source code-based workload generation requires estimating the amount of elementary operations by examining the function source codes line by line. This approach results in quite detailed and accurate workload models but requires a significantly high amount of manual work.

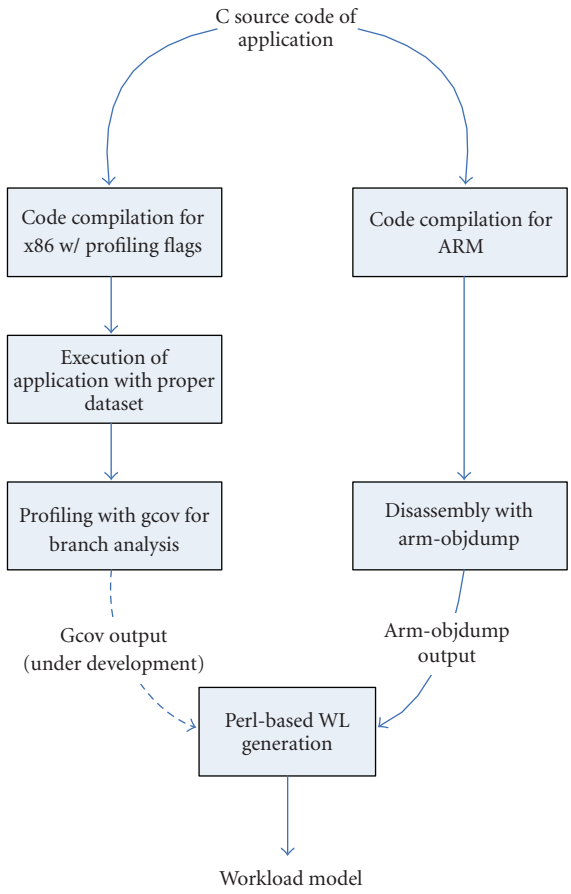


Figure 10: Semiautomatic method for C source code-based load information extraction.

A semiautomatic tool has been developed for extracting the load information from C language source code (Figure 10). The use of this tool requires that the source code is available and mature enough that it can be compiled and executed.

Firstly, the source code is compiled for the ARM instruction set with a GNU GCC cross-compiler. The resulting binary is disassembled with arm-objdump, which reveals the instructions the program would execute on an ARM processor. This arm-objdump output is then analysed by a perl script, which produces abstract instructions for the load primitive workload layer from the dump. Each load and store instruction in the dump will result to a corresponding read or write primitive and all the other instructions are replaced with execute primitives. It should be noted that the ARM compiler is used because it is easier to differentiate between memory access and other instructions with the ARM instruction set than with, for example, x86 instruction set. The extraction method, however, is not limited to ARM.

Secondly, the application is compiled for the host architecture—typically an x86-based PC-using the native GCC tool chain. Profiling flags are enabled during the build process and the resulting binary is executed with a suitable dataset. The profiling information reveals branch probabilities and the number of times loops have been executed. The function layer of the workload models can be constructed based on this information, if necessary. For example, consider a situation where the profile reveals that one code branch is executed with a 60% probability. We can add a test based on the random number generator that results in the corresponding load primitives being executed just as often. However, combining the profiling information and the load primitive information from the ARM disassembly is done manually at the moment, though an automatic tool is under development.

The source code-based approach has been used in the modelling of a 3D game [23], MP3 player, [24] and MPEG4 video encoder [19]. It has also been used partially in a case consisting of internet browsing over virtual network computing (VNC) session on a handheld device. Apart from the 3D game, these case examples have also been verified by executing the same applications in a real platform. The average difference between simulations and measurements across all those cases was about 10%.

5. PERFORMANCE SIMULATION

The combined system model is built in a GNU/Linux environment for performance simulation, using GNU compiler collection (GCC), CMake cross-platform build system, and OSCI SystemC library. Currently manually created CMakeFiles are used for configuring the build. The simulator is a command line program, which is executed in a terminal window. During the simulation it prints progress information to standard output and after the simulation is complete it displays the collected performance results.

A C++- and XML-based automatic configuration tool called VTT_COGNAC is being developed to simplify the process. With this tool also the platform model is sketched in the Tau UML2 tool from a component library (like the skeleton model of the platform currently) and the parameters of the components are set up in UML. The tool reads the XML output from Tau and generates the system model by combining the SystemC workload models generated from UML and the SystemC component models in the component library. The system simulator VTT_BEER then executes the simulation.

During the simulation of the system model the workloads send load primitives and service calls to the platform model (Figure 8). The platform model processes the primitives and service calls, advancing the simulation time while doing so. The simulation run will continue until the top-level workload model stops it when the use case has been completed.

The platform model is instrumented with counters, timers, and probes, which record the status of the components during the simulation. These performance probes are manually inserted in the component models where appropriate and are flexible so that they can be used to gather information about platform performance as needed.

Typically,

(i) status probes collect information about utilisation of components and scheduling of processes performed by the operating system models,

(ii) counters are used to calculate the number of load primitives, service calls, requests, and responses performed by the components,

(iii) timers keep track of the task switch times of the OS models and processing times of services.

Once the simulation is complete, the performance probes output the collected performance data to the standard output. A C++-based tool VTT_VODKA has been developed for viewing the data, for example, processor utilisation curves, graphically. The data can be analysed and feedback given to application or platform design. For example, if the utilisation of components is low, lowering the clock frequency can be proposed to platform designers for decreasing power consumption.

In a mobile video player case [20], the simulation speed was one tenth of real time. In other words, simulating the system for one second took about ten seconds in a Linux PC with a dual-core Intel Xeon processor, although the simulation was using only one of the cores. This is faster than cycle-accurate instruction set simulation, even if the models used in the case example were not optimised. It is also fast enough for performing early-phase performance evaluation and for simulating multiple, alternative use cases for architecture exploration.

6. MOBILE VIDEO PLAYER CASE STUDY

The performance modelling approach has been applied to the mobile video player case study. In this use case, a mobile terminal user wants to view a movie on the device. He selects a movie from a list of movies available on the mobile terminal. The execution platform of the mobile terminal is assumed to provide services for storing of movie files, for playing and displaying the selected movie, and for running the application. The use case does not focus on the human-terminal interface, but on the modelling of functionalities, services, communication, and so on that execute on the platform in fulfilling the user request.

Figure 11 displays an overall view of the workload models of the case study. PlayerApplication is the model of the software application that provides a graphical interface to the user. It responds to events coming from the user and sends messages accordingly to the VideoPlayer, ObjectServer, and DisplayServer. Since our method does not simulate application functionality, the user events are either collected to an input file or hard-coded to the model. The ObjectServer manages stored video files, provides the list of video clips available for viewing to the PlayerApplication, and streams video data to the VideoPlayer when viewing files. The VideoPlayer handles video preprocessing, decoding, and postprocessing. Finally, the DisplayServer gets the uncompressed video stream from the VideoPlayer and displays it on a screen as requested by the PlayerApplication.

Figure 12 presents the application workload model of the PlayerApplication as a composite structure diagram. It consists of process control and several process workloads, namely ObjectServer, VideoPlayer, and DisplayServer processes. The VideoPlayer process workload is further decomposed in Figure 13. In a similar manner it is built of function control and several function workloads.

Figure 14 shows the video decoding/displaying control sequence inside the VideoPlayer process workload as a state machine diagram. Firstly, a handle is obtained to the video file which has been selected, then segments of the file are requested in a loop until the last segment has been received. Each segment is decoded once it has been received.

The UML-based workload models were transformed to SystemC using the code generator developed at Lund University [21]. The SystemC-based version of the transition from wf_decode_cnf to wf_postprocess_cnf state in Figure 14 is presented in Figure 15 as an example.

The platform in the mobile video player case consists of four subsystems depicted as a block diagram in Figure 16:

(i) general purpose (GP) subsystem, which is used for executing an operating system and generic applications and services, for example,

(ii) image (IM) subsystem, which is intended for hardware-accelerated image processing and video playback and recording,

(iii) storage (ST) subsystem, which contains a repository for video clips and services for loading and storing them,

(iv) display (DP) subsystem, which takes care of displaying framebuffer data on a screen, including the viewed video files.

The subsystems are interconnected by a network using a ring topology with best-effort (BE) and guaranteed throughput (GT) routing approaches. GT is used for transferring streaming data and BE is used for transferring control messages between subsystems.

The general purpose subsystem has two ARM11 general purpose processors for executing the operating system and applications. There is also a subsystem-local SDRAM memory controller and memory to be used by the two processors. For communication with the other subsystems, the GP subsystem—like all the other subsystems—has a network interface.

Image subsystem is built around the video accelerator, which provides hardware accelerated compression and decompression, preprocessing and postprocessing services. The services provided by the subsystem are controlled by a simple ARM7 general purpose processor. There is also some SRAM memory for the ARM, a video accelerator and a DMA controller for offloading large data transfers between subsystems.

Storage and display subsystems are mostly similar to the image subsystem and they contain a simple ARM, a DMA controller and a network interface. However, instead of the video accelerator the storage subsystem has local memory for

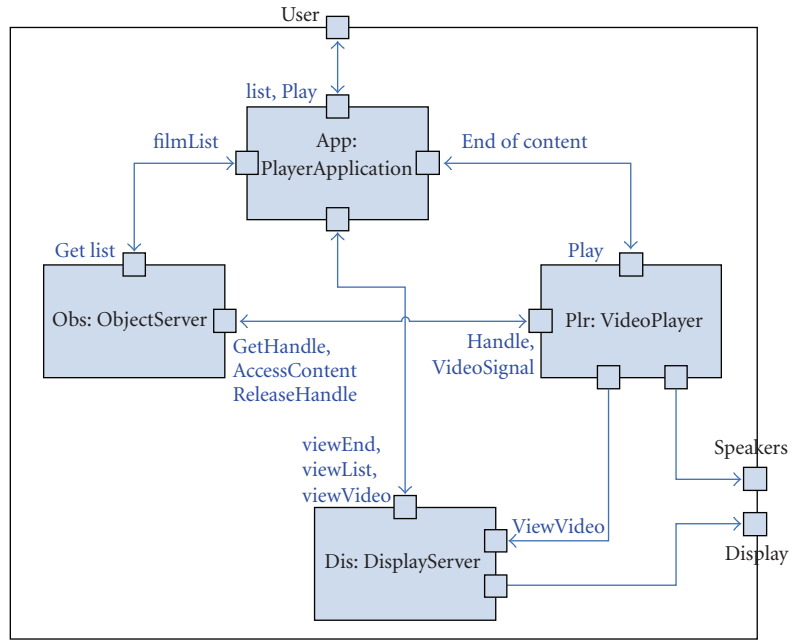


Figure 11: Interaction of the mobile video player workload models.

storage and metadata. The display subsystem has a graphics accelerator, local SRAM for graphics, and a display interface for the screen.

The platform model was created using SystemC and the communication interfaces were based on OCP TL3 on the platform layer. OCP TL2 was used on the subsystem and component layers. After the workload and platform modelling phases the workload models were mapped to the skeleton model of the platform (Figure 17). The flow of operation after the mapping is the following:

(i) the PlayerApplication, executed in one of the ARM11 processors of the GP subsystem, is launched and requests a list of movie files from the ObjectServer in the ST subsystem,

(ii) one of the movie files is selected, after which the PlayerApplication triggers the VideoPlayer in the IM subsystem,

(iii) the VideoPlayer requests the movie file from the ObjectServer, which initiates streaming of the file,

(iv) the VideoPlayer decodes the compressed stream and transfers video frames to the DisplayServer in the DP subsystem,

(v) the DisplayServer displays incoming video frames on the screen.

The video player case was simulated in the SystemC domain. The platform model was instrumented in order to collect information of the platform's performance during the simulation run. For example, the processor models record the time spent in idle, data processing, and memory accessing states for the calculation of processor utilisation. Subsystem models measure the average, minimum and maximum processing time for each service.

Table 3: Utilisation of the display subsystem components in the mobile video player use case.

Component	Idle	Busy
ARM7 microcontroller	98%	2%
SRAM controller	83%	17%
DMA controller	65%	35%
Display controller	74%	26%
Bus	66%	34%
Network interface	66%	34%

Table 4: Examples of average, minimum, and maximum processing times of services.

Service	Average	Min	Max
DMA (ST)	450 μs	1 μs	480 μs
Decoding (IM)	16 μs	14 μs	16 μs
DMA (DP)	1.4 ms	1 ms	1.6 ms

Table 3 presents the utilisation of each of the components in the display subsystem, which was obtained from the simulation. None of these components has been at the limit of their capacity: the DMA controller has been the most burdened of all and its average load has been only 35%. Furthermore, the load of the simple ARM7 microcontroller was under 3%. It is clearly possible to execute more demanding applications on this platform—at least from the display subsystem point of view. Another alternative is to reduce the clock frequency of these components to decrease the power consumption of the device.

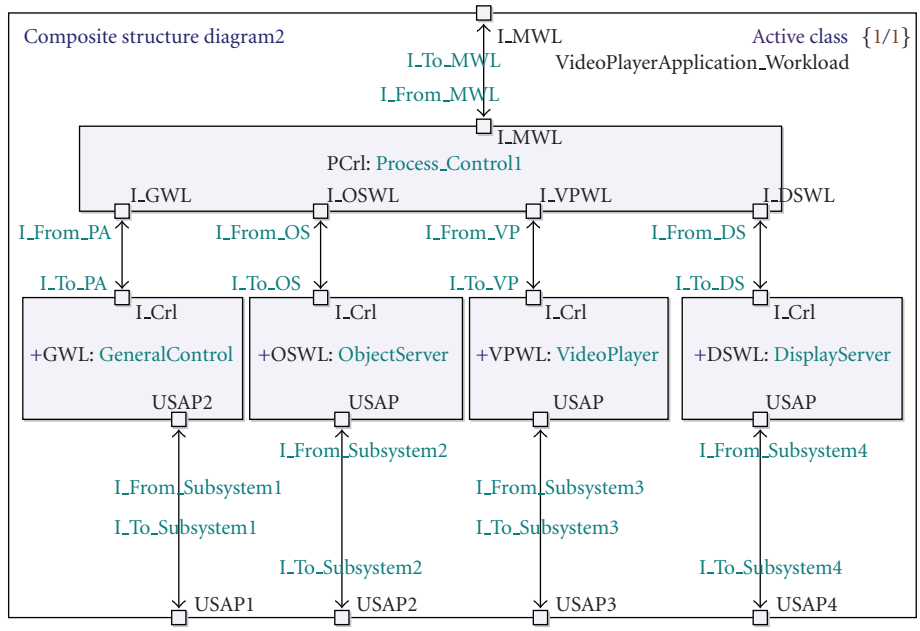


Figure 12: PlayerApplication application workload consists of process control and ObjectServer, VideoPlayer, and DisplayServer process workloads.

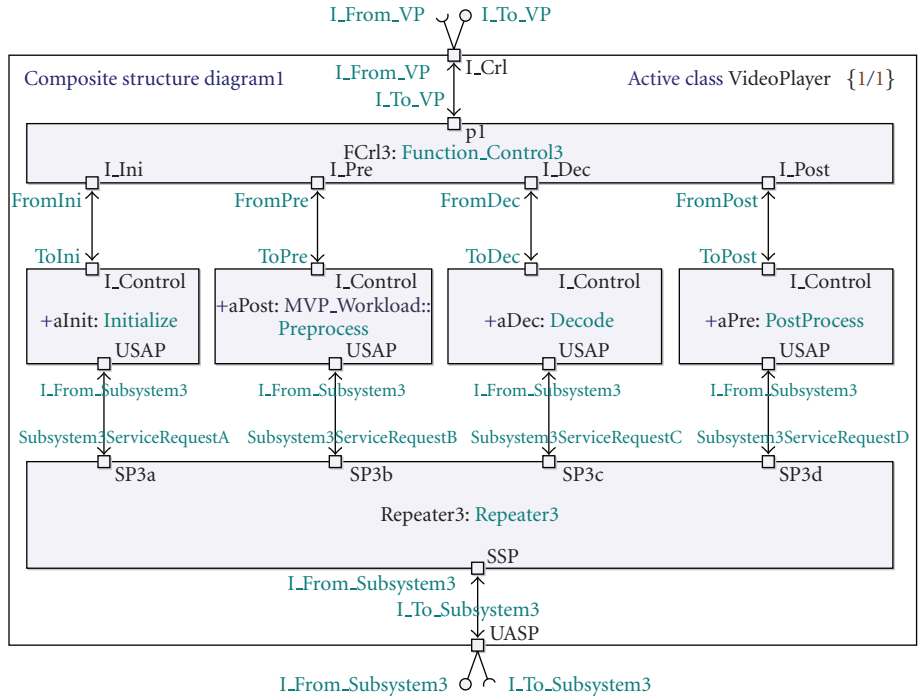


Figure 13: VideoPlayer process workload consists of function control and several functions workloads.

Another view on component utilisation is given in Figure 18, which depicts how the utilisation changes during the simulation with respect to time. The display subsystem is constantly updating the screen while the device is powered and those updates cause the lower, about 30% and 70% peaks to the utilisation curves of bus and memory components. In this simulation the video player application has been running only for a short period of time, inflicting the higher peaks.

Table 5 visualises the data reads and writes initiated or serviced by each component in the same subsystem. For example, 1.0 million read requests were processed by the bus resulting to about 4.2 million transferred 32-bit words.

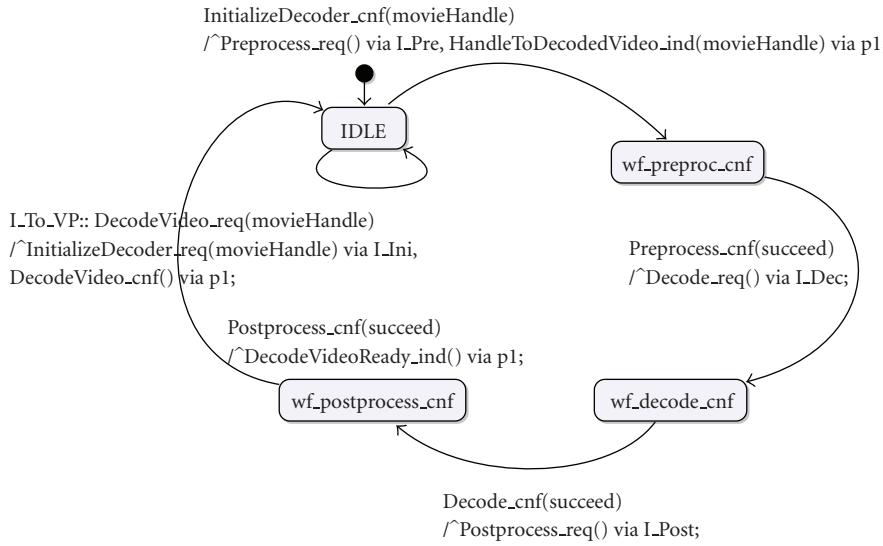


Figure 14: State machine example from the mobile video player UML model.

```
bool
MVP_Workload: :Function_Control3: :__11_trans_2
(UML_signal *pMsg)
{
  if (typeid(*pMsg) ==
      typeid(MVP_workload:: FromDec:: Decode_cnf_signal))
  {
    FromDec: :Decode_cnf_signal *pSig =
        dynamic_cast<MVP_workload:: FromDec::Decode_cnf_signal* > (pMsg);
    succeed_sm_11 = pSig → par0;
    {
      I_Post_port → Postprocess_req();
    }
    delete nextState_sm_11;
    nextState_sm_11 = new _sm_11_wf_postprocess_cnf(this);
    nextState_sm_11 → enter_action();
    return true;
  }
    return false;
}
```

Figure 15: An extract from the SystemC workload generated from the state machine diagram of Figure 14.

Table 5: Data traffic initiated or serviced by the display subsystem components.

Component	Reads	Read words	Writes	Written words
ARM7 microcontroller	1400	1400	1250	1250
SRAM controller	312 k	1245 k	635 k	2536 k
DMA controller	634 k	2534 k	634 k	2534 k
Display controller	311 k	1244 k	0	0
Bus	946 k	3780 k	635 k	2536 k
Network interface	634 k	2535 k	0	0

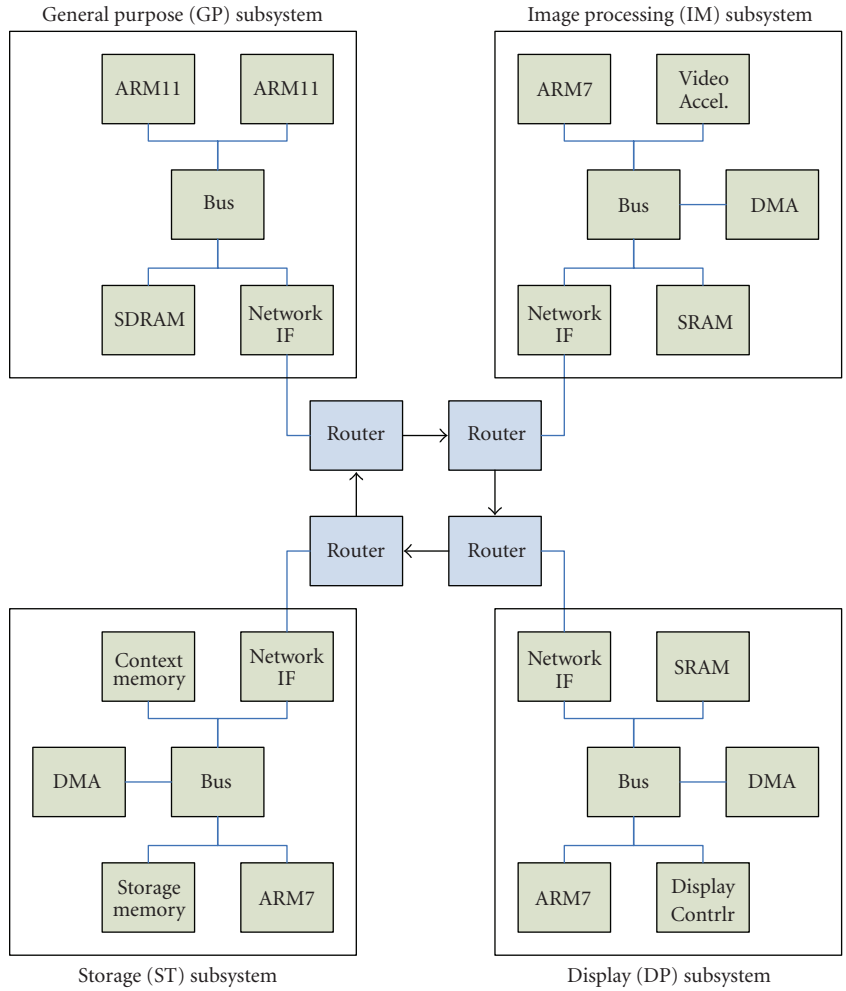


Figure 16: The execution platform of the mobile video player case consists of four subsystems.

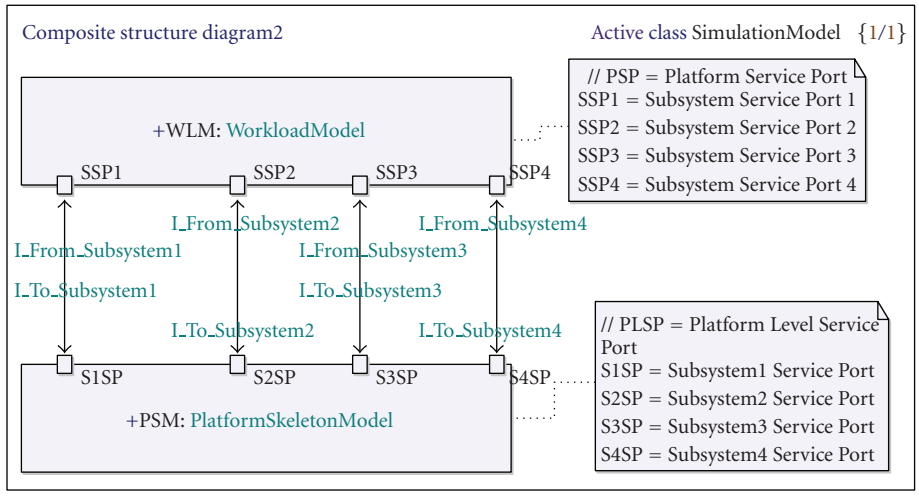


Figure 17: Mapping of the workload models to the platform skeleton model.

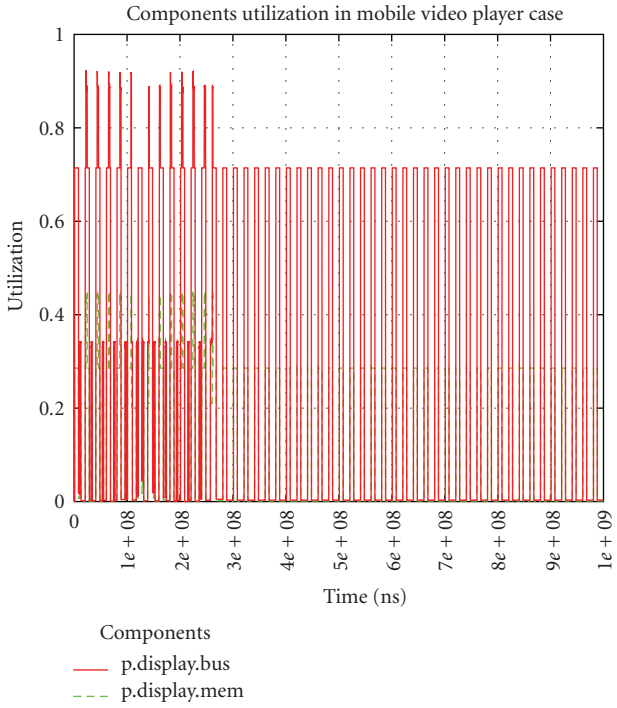


FIGURE 18: The red (upper) and green (lower) curves display the utilisation of bus and memory components in the display subsystem, respectively.

The same information as displayed in Tables 3 and 4 and Figure 18 was also collected from all the other subsystems, though they are not displayed here. Finally, Table 4 contains the processing times of, for instance, DMA transfer services from the subsystem layer.

The simulation results have not been validated with, for example, measurements since the execution platform is an invented platform intended to portray a future architecture. As such, there are no cycle-accurate simulators for the platform which could be used to obtain reference performance data. However, we have modelled MPEG4 video processing and the OMAP platform earlier and compared those simulation results to measurements from a real application in a real architecture [19]. In that case the average difference between simulations and measurements was about 12%. Furthermore, the accuracy of the simulation approach has been validated with other case examples in [23, 24].

7. CONCLUSIONS

A layered UML application/workload and SystemC platform modelling approach for performance modelling and evaluation was described. It allows application and platform to be modelled at several levels of abstraction to enable early performance evaluation of the resulting system. The approach applies a Y-chart-like specify-explore-refine paradigm. Applications are modelled in either UML or SystemC domain as workloads consisting of load primitives. Platform models are cycle-approximate transaction-level SystemC models. Mapping between UML application models and the SystemC platform models is based on automatic generation of simulation models for system-level performance evaluation.

The workload models reflect accurately the control structures of the applications. The layered hierarchical structure covers application, process, function, and basic block layers. Computation and communication are abstracted as loads that can be extracted using either analytical, measurement-based or source code-based methods, and support tools.

The platform model is an abstracted hierarchical representation of the actual platform architecture. It contains cycle-approximate timing information along with structural and behavioural aspects. The platform model is composed of three layers: component layer, subsystem layer, and platform architecture layer. Each layer has its own services, which are abstraction views of the architecture models. Services in the subsystem and platform architecture layers are invoked by workload models.

The tool support is based on a commercial UML2 tool, Telelogic Tau G2, and an open source SystemC simulation tool of OSCI. The overall performance modelling and evaluation method with support tools is called VTT_ABSOLUT2.2%. The source code-compilation method for load extraction is supported by VTT_ABSINTH. The execution platform model is configured from a library of performance models using VTT_COGNAC. The simulator is based on the open source OSCI SystemC simulator extended with configurable instrumentation and is called VTT_BEER. The simulation results can be selected for analysis and viewed using VTT_VODKA.

The overall approach has been experimented with a mobile video player case study. Unfortunately comparisons were not possible since no reference implementation was available. Different load extraction methods have been validated by applying them to MPEG-4 encoder, Quake2 3D game, and MP3 decoder case studies previously, where the average and maximum errors between simulated and monitored results have been about 15% and 25%, respectively.

The approach enables early performance evaluation, exhibits light modelling effort, allows fast exploration iteration, and reuses application and platform models. Furthermore, it provides performance results that are accurate enough for system-level exploration.

In the future, the approach will be expanded, so that power consumption or other criteria besides performance can be evaluated. As mentioned earlier, further tool support for automation of some steps of the approach is in progress. Future work will also include a real-scale case study to further validate the approach.

ACKNOWLEDGMENTS

This work is supported by Tekes (Finnish Funding Agency for Technology and Innovation), VTT under the EUREKA/ITEA contract 04006 MARTES, and partially by the European Community under the Grant agreement 215244 MOSART.

REFERENCES

[1] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "Approach for quantitative analysis of application-specific dataflow architectures," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '97)*, pp. 338–349, Zurich, Switzerland, July 1997.

[2] NoTA World Open Architecture Initiative, http://www.notaworld.org/.

[3] Open SystemC Initiative, "IEEE Standard SystemC Language Reference Manual," IEEE Computer Society, 2006. IEEE Std 1666–2005.

[4] P. Lieverse, P. van der Wolf, K. Vissers, and E. Deprettere, "A methodology for architecture exploration of heterogeneous signal processing systems," *The Journal of VLSI Signal Processing*, vol. 29, no. 3, pp. 197–207, 2001.

[5] E. A. de Kock, G. Essink, W. J. M. Smits, et al., "YAPI: application modeling for signal processing systems," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 402–405, Los Angeles, Calif, USA, June 2000.

[6] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–111, 2006.

[7] T. Wild, A. Herkersdorf, and G.-Y. Lee, "TAPES—trace-based architecture performance evaluation with SystemC," *Design Automation for Embedded Systems*, vol. 10, no. 2-3, pp. 157–179, 2005.

[8] J. M. Paul, D. E. Thomas, and A. S. Cassidy, "High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 3, pp. 431–461, 2005.

[9] D. Gajski, J. Zhu, R. Dömer, A. Gestlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[10] F. Ghenassia, Ed., *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*, Springer, New York, NY, USA, 2005.

[11] TLM2 Whitepaper, http://www.systemc.org/members/download_files/check_file?agreement=tlm2_whitepaper.

[12] H. Posadas, F. Herrera, P. Sánchez, E. Villar, and F. Blasco, "System-level performance analysis in SystemC," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 1, pp. 378–383, Paris, France, February 2004.

[13] T. Kangas, P. Kukkala, H. Orsila, et al., "UML-based multiprocessor SoC design framework," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 281–320, 2006.

[14] Ptolemy II—heterogeneous modelling and design, http://ptolemy.berkeley.edu/ptolemyII.

[15] J. Eker, J. W. Janneck, E. A. Lee, et al., "Taming heterogeneity—the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–143, 2003.

[16] A. L. Sangiovanni-Vincentelli, "Quo vadis SLD: reasoning about trends and challenges of system-level design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[17] UML Profile for MARTE Beta 1, http://www.omg.org/cgi-bin/doc?ptc/2007-08-04.

[18] J. Kreku, Y. Qu, J.-P. Soininen, and K. Tiensyrjä, "Layered UML workload and SystemC platform models for performance simulation," in *Proceedings of the International Forum on Specification and Design Languages (FDL '06)*, pp. 223–228, Darmstadt, Germany, September 2006.

[19] J. Kreku, M. Eteläperä, and J.-P. Soininen, "Exploitation of UML 2.0-based platform service model and SystemC workload simulation in MPEG-4 partitioning," in *Proceedings of International Symposium on System-on-Chip (SoC '05)*, pp. 167–170, Tampere, Finland, November 2005.

[20] J. Kreku, M. Hoppari, K. Tiensyrjä, and P. Andersson, "SystemC workload model generation from UML for performance simulation," in *Proceedings of the International Forum on Specification and Design Languages (FDL '07)*, Barcelona, Spain, September 2007.

[21] P. Andersson and M. Höst, "UML and SystemC-comparison and mapping rules for automatic code generation," in *Proceedings of the International Forum on Specification and Design Languages (FDL '07)*, Barcelona, Spain, September 2007.

[22] P. Andersson, M. Höst, and M. Bengtström, "UML to SystemC transformation in the MARTES project," in *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA '06)*, Cavat/Dubrovnik, Croatia, August-September 2006.

[23] J. Kreku, T. Kauppi, and J.-P. Soininen, "Evaluation of platform architecture performance using abstract instruction-level workload models," in *Proceedings of International Symposium on System-on-Chip (SoC '04)*, pp. 43–48, Tampere, Finland, November 2004.

[24] J. Kreku, J. Penttilä, J.-P. Soininen, and J. Kangas, "Workload simulation method for evaluation of application feasibility in a mobile multiprocessor platform," in *Proceedings of the Euromicro Symposium on Digital System Design (DSD '04)*, pp. 532–539, Rennes, France, August-September 2004.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 726096, 15 pages
doi:10.1155/2008/726096

Research Article

Automated Integration of Dedicated Hardwired IP Cores in Heterogeneous MPSoCs Designed with ESPAM

Hristo Nikolov, Todor Stefanov, and Ed Deprettere

Leiden Embedded Research Center, Leiden Institute of Advanced Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands

Correspondence should be addressed to Hristo Nikolov, nikolov@liacs.nl

Received 27 September 2007; Revised 20 February 2008; Accepted 7 April 2008

Recommended by Sandeep Shukla

This paper presents a methodology and techniques for automated integration of dedicated hardwired (HW) IP cores into heterogeneous multiprocessor systems. We propose an IP core integration approach based on an HW module generation that consists of a wrapper around a predefined IP core. This approach has been implemented in a tool called ESPAM for automated multiprocessor system design, programming, and implementation. In order to keep high performance of the integrated IP cores, the structure of the IP core wrapper is devised in a way that adequately represents and efficiently implements the main characteristics of the formal model of computation, namely, Kahn process networks, we use as an underlying programming model in ESPAM. We present details about the structure of the HW module, the supported types of IP cores, and the minimum interfaces these IP cores have to provide in order to allow automated integration in heterogeneous multiprocessor systems generated by ESPAM. The ESPAM design flow, the multiprocessor platforms we consider, and the underlying programming (KPN) model are introduced as well. Furthermore, we present the efficiency of our approach by applying our methodology and ESPAM tool to automatically generate, implement, and program heterogeneous multiprocessor systems that integrate dedicated IP cores and execute real-life applications.

1. INTRODUCTION

For modern embedded systems in the realm of high-throughput multimedia, imaging, and signal processing, the complexity of embedded applications has reached a point where the performance requirements of these applications can no longer be supported by embedded system architectures based on a single processor. Thus the emerging embedded system-on-chip platforms are increasingly becoming multiprocessor architectures (MPSoCs). There are several possibilities in building MPSoCs.

(1) *Homogeneous MPSoCs.* In these platforms, each processing component is a fixed ISA programmable processor, for example, IBM's *PowerPC* 405 [1], XILINX's *MicroBlaze* [2], and so on. The advantage of such systems is that in order to generate an executable code, a standard (e.g., C/C++) compiler is used. This leads to high flexibility in case different applications have to be mapped on a particular multiprocessor platform or a new version of an application is to be deployed on the same platform. However, the overhead in the executable code introduced by the compiler in general limits the performance these systems can achieve.

(2) *Multi-ASIP SoCs.* The advances in the *application specific instruction-set processors (ASIPs)* technology [3] raise an alternative to build high-performance systems. The ASIPs extend the fixed ISA processors to the so-called (re-)configurable or extensible processors. Examples of this approach are Xtensa LX configurable processor from (Tensilica, Santa Clara, Calif, USA) [4], IXP1200 network processor from (Intel, Santa Clara, Calif, USA) [5], and TriMedia TM3270 media processor from (NXP, Eindhoven, The Netherlands) [6]. The ASIPs approach enables system designers to add application- (domain) specific extensions (instructions) to a base processor that may have never been considered or imagined by designers of the original architecture. The addition of highly customized instructions matched perfectly to a specific application gives ASIPs the ability to deliver performance higher than the conventional fixed ISA

processors. As a consequence, building multi-ASIP systems would give better performance than MPSoCs utilizing fixed ISA processors. Although several design methodologies for building ASIPs exist [7, 8], building a multi-ASIP system is yet far from trivial and faces the same problems as any multiprocessor system, that is, synchronizing the interprocessor data communications and programming the system as a whole (more sophisticated parallelizing compilers are required). Moreover, to efficiently customize a multi-ASIP platform, additional design effort and tools are required.

(3) *Heterogeneous MPSoCs consisting of programmable processors and dedicated HW IP cores.* The *programmable* processors in such MPSoCs can be both fixed ISA and configurable (ASIP) processors. For efficiency, in a multiprocessor system different tasks have to be executed by different types of processing components which are optimized for execution of particular tasks. It is a common knowledge that higher performance is achieved by a dedicated (customized and optimized) HW IP core because it works more efficiently than fixed ISA and ASIP processors. Moreover, many companies already provide dedicated customizable IP cores optimized for a particular functionality that aim at saving design time and increasing overall system performance. Therefore, the idea of using dedicated IP cores in heterogeneous systems is very appealing. These heterogeneous systems are very attractive because they deliver high flexibility and high performance at the same time.

The ever increasing time-to-market pressure requires for systematic and, moreover, automated design methodologies for building MPSoCs where flexibility and IP reuse are very important aspects. However, two major problems emerge, namely, how to design, that is, how to integrate different dedicated HW IP cores and programmable processors into heterogeneous multiprocessor systems, and how to program these systems. The lack of standard interfaces that an IP core has to provide in order to allow seamless integration, and the lack of automated programming approaches for heterogeneous multiprocessor systems only exacerbates these problems.

1.1. Paper contributions

As a particular solution to the problems stated above, in this paper, we present our method and techniques for automated generation of *heterogeneous* multiprocessor systems where both fixed ISA processors and dedicated HW IP cores are used as processing components. Currently, ASIPs are out of the scope of this work because integrating ASIPs together with fixed ISA processors and HW IP cores adds an extra dimension in the complexity of the problems. Our approach is to solve the problems gradually, therefore, in this work, we investigated only the integration of third-party HW IP cores with fixed ISA processors. Our approach for integrating dedicated IP cores into a system is based on an automated HW module generation consisting of a wrapper around a predefined IP core. The structured, highly modularized, parameterized, and efficient design of our HW module is one of the contributions of this paper. This contribution substantially extends the embedded system-level platform synthesis and application mapping (ESPAM) tool [9] and allows automated generation and implementation of heterogeneous multiprocessor systems.

Another important contribution is that with ESPAM a heterogeneous multiprocessor system is programmed automatically where a part of the input application, initially specified as a sequential C program, is executed by programmable processors (ESPAM generates program code for each processor), and a part is implemented by dedicated IP cores. For the latter ESPAM integrates them by generating HW modules and connecting them to the other processing components of the system. Moreover, the structure of the HW module we propose allows an efficient integration and connection to alternative communication structures supported by ESPAM without the need of any additional modifications of the IP cores.

The success of our method and techniques for automated programming of the processors in a multiprocessor system and automated generation of HW modules for IP core integration into heterogeneous systems is based on the underlying programming (application) model used in ESPAM. ESPAM targets data-flow dominated (streaming) applications for which we use the Kahn process network (KPN) [10] model of computation. Many researchers [11–17] have already indicated that KPNs are suitable for efficient mapping onto multiprocessor platforms. However, decomposing an application into a set of concurrent tasks is one of the most time-consuming jobs imaginable [18]. Fortunately, our tool flow and programming approach combines a tool called PNgen [19] that we have developed for automatic derivation of KPN specifications from applications specified as sequential C programs.

The structure of the HW module we propose has been devised by carefully exploiting and efficiently implementing the simple communication and synchronization mechanisms of the KPN model. We have identified and developed a library, that is, a set of generic parameterized components used by ESPAM to compose an HW module. This is done by instantiating components from the library, connecting them, and setting their parameters in correspondence with the KPN specification. We consider our library of generic components as an important contribution because by making the HW module clearly structured and modularized, every component becomes more independent and loosely coupled. Therefore, we are able to design and optimize each component of the HW module separately. This brings much convenience for efficient and effective optimization, making the performance of the generated systems better.

Finally, we would like to mention that the ESPAM and PNgen tools are open source projects, available for third parties, and can be downloaded from [20].

1.2. Scope of work

In this section, we outline the assumptions and restrictions regarding our work presented in this paper. Most of them are discussed in detail, where appropriate, throughout the paper.

1.2.1. Applications

One of the main assumptions for our work is that we consider only data-flow dominated applications in the realm of multimedia, imaging, and signal processing that naturally contain tasks communicating via streams of data. Such applications are very well modeled by using the parallel data-flow model of computation called Kahn process network (KPN). The KPN model we use is a network of concurrent autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels, using a blocking read/write on an empty/full FIFO as a synchronization mechanism. Furthermore, we consider KPNs that are input-output equivalent to static affine nested loop programs (SANLP). The properties of such programs are discussed in Section 1.2.5. We are interested in this subset of KPNs because they are analyzable at compile time (e.g., FIFO buffer sizes and execution schedules are decidable) and, as we show in this paper, HW synthesis from them is possible. Moreover, such KPNs can be derived automatically from the corresponding sequential programs [19, 21–23].

1.2.2. FIFO-based integration of IP cores

A key feature of our IP integration methodology is that the IPs are wrapped with a HW module and the communication is enabled through FIFOs. In order to achieve high performance with low overhead, our FIFO-based integration approach strictly follows the semantics of the KPN model of computation. Therefore, we motivate this approach by explaining the most favorable characteristics of the KPN model that we exploit in order to allow seamless and automated IP integration.

(1) The KPN model is determinate, which means that irrespective of the schedule chosen to run the network, the same input/output relation always exists. This gives a lot of scheduling freedom that can be exploited when integrating many different hardware IP cores in a heterogeneous multiprocessor system.

(2) The interprocess synchronization in a KPN is done by a blocking read/write on empty/full FIFOs. This is a very simple synchronization protocol that can be realized easily and efficiently in hardware, thus making the synchronization, which is an essential part of our IP integration approach, very simple and efficient in terms of utilized hardware.

(3) The control in a KPN is completely distributed to the individual processes. Therefore, there is no global scheduler present. As a consequence, the integration of many hardware IP cores in a multiprocessor system is a simple task from the point of view that the local IP controllers do not have to be connected to and interact with a global scheduler controller.

(4) The exchange of data among processes in a KPN is distributed over the FIFOs. There is no notion of a single global data memory that has to be accessed by multiple processes. Therefore, when multiple IP cores are integrated into a multiprocessor system and communicate data via distributed FIFO buffers, resource contention is greatly reduced.

1.2.3. Multiprocessor platforms

We consider multiprocessor platforms in which the processing components, that is, programmable processors and/or HW IP cores, communicate data only through distributed memory units. Each memory unit can be organized as one or several FIFOs. The data communication among the processing components is realized by blocking read and write synchronization mechanism. Such platforms match and support very well the KPN operational semantics, thereby achieving high performance when KPNs are executed on the platforms. Also, compliant with the operational semantics of a KPN, our platforms support blocking synchronization mechanism, allowing the processors to be self-scheduled, avoiding a global scheduler component. If the number of processing components in a platform is less than the number of processes of a KPN, then some of the programmable processors execute more than one process. ESPAM schedules these processes at compile time and generates program code for a given processor which code does not require/utilize an operating system. In our approach, a HW IP core implements the computation of a single-KPN process. Therefore, we do not support more than one KPN process to be implemented by a single-HW IP core. Additional requirements for the HW IP cores are discussed in Section 3.2. The programmable processors and the HW IP cores in our platforms can be connected by a crossbar switch, a point-to-point network, or a shared bus. Additional details are given in Section 2.2.

1.2.4. Research and tools

The KPN model of computation has been widely studied in our group for more than 7 years. This research resulted in techniques and tools implemented in the Compaan/Laura design flow [11] for automated translation of SANLPs written in Matlab [21, 23] to KPN specifications targeting dedicated HW implementations on FPGAs [24]. Although these techniques are very advanced, they currently generate KPNs with too many FIFO channels which may lead to inefficient implementations. Also, they do not address the problem of what the FIFO buffer sizes should be. This is a very important problem because if the FIFO buffers are undersized, this leads to a deadlock in the KPN behavior.

We addressed the problems above, and based on the knowledge we have obtained working on Compaan, recently we have developed techniques for *improved* derivation of KPNs implemented in the PNgen tool [19]. These techniques allow for automated computation of efficient buffer sizes of the communication FIFO channels that guarantee deadlock-free execution of our KPNs. In addition, our group started a research on design automation for high-performance multiprocessor systems, a challenging and very appealing domain nowadays. We devised a novel approach for automated *homogeneous* multiprocessor systems design, programming, and implementation. The concept and the techniques behind it were implemented in a tool called ESPAM [9]. Until recently, our approach was limited in the sense that only *homogeneous* systems containing programmable processors

could be designed with ESPAM. Consequently, based on our experience with Laura [24], we developed techniques that substantially extend the functionality and flexibility of ESPAM allowing automated integration of programmable processors and dedicated IP cores into heterogeneous systems. As we mentioned, the IP core integration is based on generation of a HW module consisting of a wrapper around the IP. This approach originates from the general idea implemented in Laura, that is, generating of HW modules based on the properties of the KPN model we use. Although using the same concept in ESPAM, we developed different techniques in order to enable automated IP cores integration and connection to the other components of the system, that is, programmable processors and different communication structures. By carefully exploiting the main features of a KPN process, we created a library of parameterized components used to compose a HW module. In addition, we defined clear interfaces of the HW module and its components. This helped us to devise an efficient mechanism for connecting and synchronizing the components within an HW module keeping high performance of the integrated IP cores.

1.2.5. Tools inputs

Our ESPAM tool accepts as an input three specifications, that is, a platform specification, a mapping specification, and an application specification. The platform specification is restricted in the sense that it must specify a platform that consists of components taken from a predefined set of components. This set ensures that many alternative multiprocessor platform instances can be constructed and all of them fall into the class of platforms we consider (see above). The mapping specification can specify one-to-one and/or many-to-one mappings (only for programmable processors) of processes onto processing components. Based on this mapping, ESPAM determines automatically the most efficient mapping of FIFO channels onto distributed memory units. The application specification is a KPN in XML format derived from an application written as a static affine nested loop program (SANLP) in C (PNgen) or Matlab (Compaan). The SANLPs are programs that can be represented in the well-known polytope model [25]. That is, an SANLP consists of a set of statements and function calls, each possibly enclosed in loops and/or guarded by conditions. The loops need not be perfectly nested. All lower and upper bounds of the loops as well as all expressions in conditions and array accesses have to be affine functions of enclosing loop iterators and parameters. The parameters are symbolic constants, that is, their values may not change during the execution of the program. The above restrictions allow a compact mathematical representation of an SANLP, enabling the development of techniques for automated KPN derivation. Many applications in the domain we consider (see above) can be represented as SANLPs.

1.3. Related work

Several frameworks exist that address the issue of IP or component integration. Some of the most important are liberty simulation environment (LSE) [26], BALBOA [27], MCF [28], and Ptolemy [29]. In general, the methodologies proposed in these frameworks are related to our ESPAM methodology in the sense that we also propose a component (IP)-based approach to heterogeneous systems design. However, a major difference is that ESPAM addresses a different aspect of the heterogeneous systems design. ESPAM is a component-based framework targeting automated system synthesis, automated programming, and automated physical implementation of heterogenous multiprocessor systems whereas liberty, BALBOA, MCF, and Ptolemy are component-based frameworks/environments for system modeling and simulation.

The *control-read-execute-write* paradigm used to implement the operational semantics of our HW module wrapper is very similar to the paradigm used in [29, 30]. The difference is that in [29, 30], an implementation of similar paradigm is presented which is suitable and can be used only for hierarchical modeling and simulation purposes. In our ESPAM environment, we show an implementation of our *control-read-execute-write* paradigm that allows IP core integration in a physical multiprocessor system. Moreover, we present how our *control-read-execute-write* implementation is fully automatically synthesized/generated from sequential C code.

There are several approaches for HW design based on the ANSI C standard such as Handel-C (commercialized by Celoxica, Abingdon, Oxfordshire, UK [31]) and SpecC [32]. In contrast to our approach for multiprocessor systems design, Handel-C targets dedicated HW implementations on FPGAs. To express parallelism and event sensitivity in Handel-C, a designer has to use annotations (construct *par*) in the programming code. SpecC is a modeling language for the specification and design of embedded systems at system level. In [32], the authors propose a design methodology based on a library of reusable components that includes several steps which is similar to our methodology. The main difference, however, is that SpecC is an extension of the C programming language implying that the designer has to study it. Also, with SpecC the designer has to specify the possible parallelism of an application in an explicit way. In contrast to Handel-C and SpecC, the application specification in our methodology is a C program written by using a subset of the ANSI C standard without any special annotations, that is, SANLP explained in Section 1.2.5. A SystemC-based approach for design automation of digital signal processing systems is presented in [33]. The proposed methodology consists of an automated design space exploration, performance evaluation, and automatic platform-based system generation. Similarly to our approach, the input for the design flow contains an executable application specification (written in SystemC), a target architecture template (in both approaches built from components taken from a component library) and mapping constraints of the SystemC modules (in our methodology, we have a mapping giving a relation between the application and the architecture). In order to automate the design process, the SystemC application has to be written in a synthesizable subset of SystemC, called SysteMoC [34], whereas our

restriction of the initial C program is to be an SANLP (see Section 1.2.5). The synthesizable subset of SystemC is required because for the IP core generation the authors use high-level synthesis tools which is a major difference with our concept for heterogeneous MPSoCs design. Instead, in this paper we propose an approach for dedicated IP core integration based on a HW module generation consisting of a wrapper around a predefined IP core.

In our automated design flow for MPSoC design, we use the KPN model of computation to represent an application and to map it onto alternative heterogeneous MPSoC architectures. A similar approach is presented in [35, 36]. Jerraya et al. propose a design flow concept that uses high-level parallel programming model to abstract HW/SW interfaces in the case of heterogeneous MPSoC design. The multiflex system presented in [37] is an application-to-platform mapping tool targeting multimedia and networking applications. Multiflex uses symmetric multiprocessing (SMP) and distributed system object component (DSOC) programming models. For implementation, the multiflex system targets the StepNP MPSoC platform architecture. Although the work presented in [35–37] targets heterogeneous MPSoC design, the authors do not address the problem of automated integration of dedicated HW IP cores into their heterogeneous MPSoCs. In contrast, in this paper we present efficient techniques for automated integration of IP cores into heterogeneous multiprocessor systems designed with ESPAM.

There are several initiatives such as VSIA [38] and OCP-IP [39] aiming at specifying "open" interface standards, for example, the virtual component interface (VCI) and the open core protocol (OCP), which will ease the integration effort required to incorporate IP cores into a system-on-chip. SPIRIT [40] is a consortium which aims at "enabling innovative IP reuse and design automation." IP-XACT is a standard defined by SPIRIT that allows to use general interfaces for connection between the IPs, where for each interface a reference bus definition is required. The main focus of these initiatives is to guarantee interoperability and reusability of a wide variety of IP cores in a "plug-and-play" fashion but this is achieved at the expense of more general, application-independent, and relatively slow interfaces and protocols. Our IP wrappers developed in ESPAM are not meant to be as general as VCI and OCP, that is, our wrappers support efficient integration of the specific type of IPs defined in Section 3.2. This fact and the fact that our wrappers are customized for every application, that is, they are automatically generated according to the KPN specification of an application, guarantee that the highest possible overall system performance is achieved.

The task transaction level (TTL) interface presented in [41] is a design technology for programming of embedded multiprocessor systems. Our programming approach is similar to TTL in the sense that both target streaming applications and both use communication primitives. However, in our approach, we consider only MPSoC architectures with distributed memory because such architectures give better timing performance compared to shared memory architectures. TTL is more flexible because it supports many communication primitives but programming an MPSoC by using TTL requires a lot of manual work which is hard (in some cases even impossible) to automate. In [10], Kahn proved that by using infinite FIFO queues, the blocking read in-order mechanism is sufficient to realize communication and synchronization in any streaming application modeled as a process network. Due to practical reasons, blocking write is needed as well because a FIFO implementation cannot have an infinite size. However, using a blocking write mechanism and finite memory resources may lead to deadlock of a KPN when executed. Therefore, we developed techniques for computing FIFO sizes such that a deadlock-free execution of our KPNs on our platforms is guaranteed [19]. In this sense, the blocking read and write, both in-order, form the minimum set of basic communication primitives realizing the communication mechanism of a process network when targeting real implementations. Other communication/synchronization mechanisms add more flexibility but at a certain price. In comparison with TTL, our platform model supports only the two basic primitives, which allows us to fully automate the programming of MPSoCs.

2. PRELIMINARIES

The paper is organized as follows. Here, in Section 2, we give an overview of our system design methodology and techniques centered around our ESPAM tool [9]. This is necessary in order to understand the main technical contribution of this paper presented in detail in Section 3, namely, automated IP core integration allowing automated heterogeneous MPSoCs generation, programming, and implementation with ESPAM. In Section 4, we present some results that we have obtained by using ESPAM to design efficient heterogeneous MPSoCs. Section 5 concludes the paper.

2.1. ESPAM design flow

Our system design methodology is depicted as a design flow in Figure 1. There are three levels of specification in the flow. They are System-Level specification, RTL-Level specification, and Gate-Level specification. The System-Level specification consists of three parts written in XML format: (1) *Platform Specification* describing the topology of a multiprocessor platform; (2) *Application Specification* describing an application as a Kahn process network (KPN) [10], that is, network of concurrent processes communicating via FIFO channels. The KPN specification reveals the task-level parallelism available in the application. In the presented design flow, we start from a sequential program written in C and the PNgen tool [19] automatically converts it into a KPN specification. (3) *Mapping Specification* describing the relation between all processes and FIFO channels in *Application Specification* and all components in *Platform Specification*. The platform and the mapping specifications can be written by hand or can be generated automatically after performing a design space exploration. For this purpose, we use the Sesame tool [14]. The System-Level specification is given as input to ESPAM. First, ESPAM constructs a platform instance from the platform description. The platform instance is

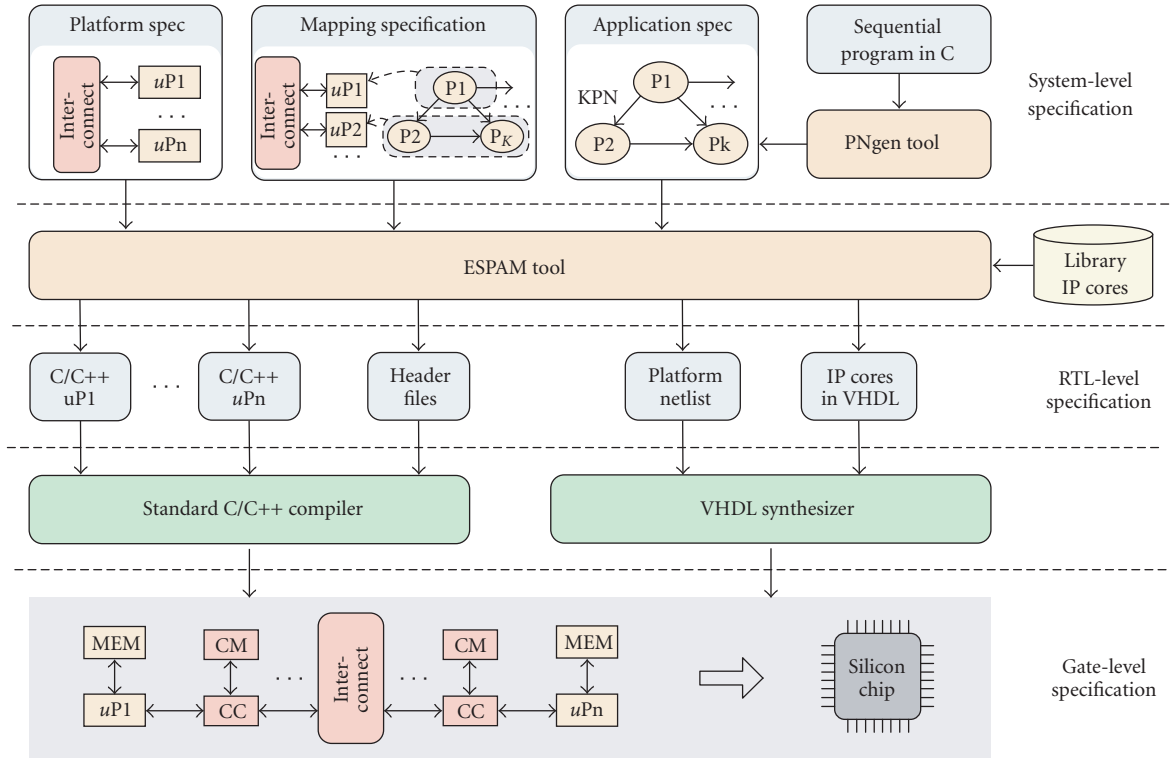


Figure 1: ESPAM system design flow.

an abstract model of an MPSoC because, at this stage, no information about the target physical platform is considered. The model defines only the key system components of the platform and their attributes. Second, ESPAM refines this abstract platform model to an elaborate parameterized RTL model (RTL-Level specification of an MPSoC) using the components from the IP library (see Figure 1). The refined system components are instantiated by setting their parameters based on the target physical platform features. Finally, ESPAM generates C/C++ program code for each processor in the multiprocessor platform in accordance with the application and mapping descriptions. The program code is further given to a standard GCC compiler to generate executable code. In addition, a commercial synthesizer converts the RTL-Level HW specification to a Gate-Level specification, thereby generating the target platform gate level netlist, see the bottom part of Figure 1. This Gate-Level specification is actually the system implementation.

2.2. Multiprocessor platforms

Our ESPAM flow presented in [9] only supports automated design of *homogeneous* multiprocessor platforms, that is, platforms containing only programmable processors. Here, we briefly describe these platforms in order to show and describe clearly, later in Section 3, how we extend ESPAM to support automated design of *heterogeneous* multiprocessor platforms. The homogeneous multiprocessor platforms considered in [9] are constructed by connecting *processing*, *memory*, and *communication* components using *communication controllers (CCs)*. Our approach is explained below using the example of a multiprocessor platform depicted in the bottom-left part of Figure 1. It contains several processors connected to a communication component (INTERCONNECT) using communication memories (CMs), and communication controllers (CCs). The processors have separate program/data memory (MEM) and transfer data between each other through the CM memories. A communication controller connects a communication memory to the data bus of the processor it belongs to and to a communication component. Each CC implements the processor's local bus-based access protocol to the CM for write operations and the access to the communication component (INTERCONNECT) for read operations. In our approach, each processor writes only to its local communication memory (CM) and uses the communication component only to read data from all other communication memories. Each CM is organized as one or more FIFO buffers. We have chosen such organization because, then, the interprocessor synchronization in the platform can be implemented in a very simple and efficient way by blocking read/write operations on empty/full FIFObuffers located in the communication memory.

The FIFO organization of the CMs is implemented by the CCs in hardware which leads to very efficient data transfer, that is, a processor writes/reads a 32-bit word to/from a FIFO in 4 clock cycles [9]. If a processor supports direct FIFO communication, for example, the XILINX's *MicroBlaze* processor [2] used in ESPAM has several dedicated FIFO (FSL) interfaces (similar to Tensilica's Xtensa LX), then a CC implements a single FIFO which is directly connected to the

(a)

```
1   // process P1
2   void main() {
3     for(int k = 1; k <= L; k + +) {
4         read(IP1, in_0, size);
5         execute(in_0, out_0);
6         write(OP1, out_0, size);
7   } }

8   void read(byte *port, void *data, int length) {
9     int *isEmpty = port +1;
10    for(int i = 0; i < length; i + +) {
11      //reading is blocked if a FIFO is empty
12      while(*isEmpty){}
13      (byte *data)[i] = *port; // read from a FIFO
14  } }

15  void writ(byte *port, void *data, int length) {
16    int *isFull = port +1;
17    for(int i = 0; i < length; i + +) {
18      //writing is blocked if a FIFO is full
19      while(*isFull){}
20      *port = (byte *data)[i]; // write to a FIFO
21  } }
```

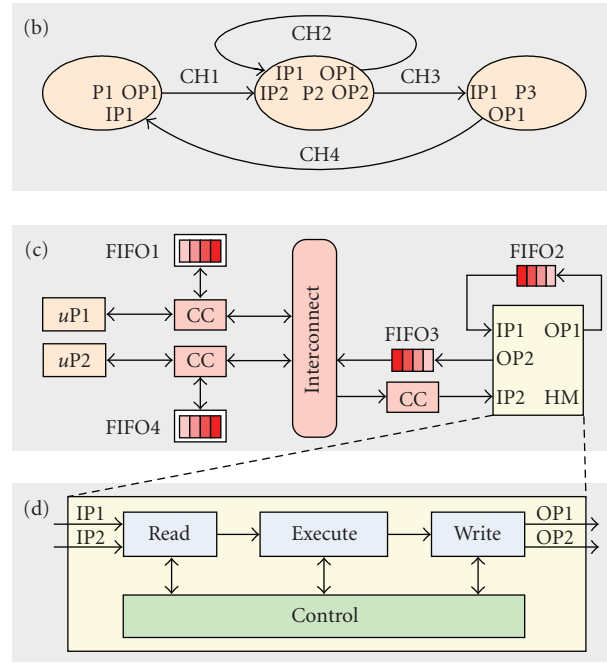


Figure 2: Example of heterogeneous MPSoC generated by ESPAM.

data path of the processor (through an FSL). In this case, the *MicroBlaze* processor uses a specific instruction to access the FIFO, reducing the transferring time to only 2 clock cycles (if blocking does not occur). The reduced time is caused by the fact that in the FSL communication, the blocking synchronization is supported and implemented by the state machine of the processor. For some applications, however, the number of FIFOs connected to a processor may exceed the number of direct FIFO links supported by the processor. In this case, ESPAM first utilizes all the available direct FIFO interfaces and then connects the remaining FIFOs to the processor data bus.

2.3. Automated programming

Our methodology and tool-flow for multiprocessor system design allow automated synthesis, programming, and implementation of multiprocessor platforms. Automated programming of an MPSoC means that the ESPAM tool automatically generates program code for each processor in the system, generates the memory map of the system, and generates code that implements the synchronization and communication between the processors. In our methodology and design flow, the first step is partitioning of an application into concurrent tasks where the intertask communication and synchronization is *explicitly* specified in each task. Such partitioning, performed automatically by the PNgen tool [19], allows each task or group of tasks to be compiled separately by a standard *C* compiler in order to generate an executable code for each processor in the platform. The partitioning of an application into concurrent tasks requires the use of a parallel model of computation in order to functionally specify the application. In our case, PNgen uses and generates such model, namely, a Kahn process network (KPN), for a given application initially specified as a sequential static affine nested loop C program. A KPN generated by PNgen is a network of concurrent autonomous processes that communicate data in a point-to-point fashion over FIFO channels, using blocking read/write on empty/full FIFOs as a synchronization mechanism. Moreover, regardless of the functional behavior specified by processes in a KPN generated by PNgen, always ESPAM takes each process specification and generates a specific code for each process where the structure of the code is the same for all processes to be executed by programmable processors. This uniform structure is explained by an example.

Consider the KPN shown in Figure 2(b). Three processes (P1, P2, and P3) are connected through four FIFO channels (CH1, CH2, CH3, and CH4). The structure of each process is the same and consists of a CONTROL part, a READ part, an EXECUTE part, and a WRITE part where the parts are specific—see, for example, process P2 in Figure 3(a). The same structure can be seen for process P1 in Figure 2(a)—see lines 2 to 7. The difference between P2 and P1 is in the specific code in each part. For example, the CONTROL part of P2 has two for-loops whereas the CONTROL part of P1 has only one for-loop. The READ part of P2 has two read primitives and if conditions specifying when to execute these primitives whereas the READ part of P1 has one read primitive executed unconditionally.

The blocking read/write synchronization mechanism of our KPNs is implemented by read/write synchronization primitives. They are the same for each process and are depicted in Figure 2(a)—see lines 8 to 21. Detailed explanation of the primitives' code can be found in [9]. The primitives are automatically generated and inserted in the

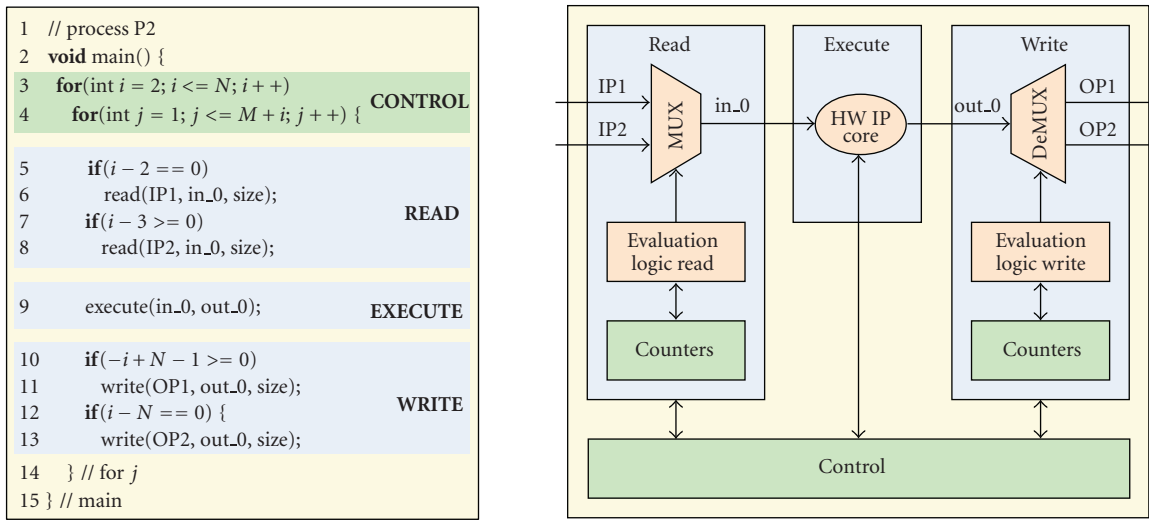


Figure 3: Example of a HW module and its blocks' structure.

program code by ESPAM in the places where a process has to read/write data from/to a FIFO channel. For example, process P1 in Figure 2(a) reads data from its input channel via port IP1 (line 4). If data is not available, then the process blocks on reading until data arrives. Then it performs a computation on the data (line 5), and writes the result to its output FIFO channel via port OP1 (line 6). If the FIFO is full, then the process blocks on writing until there is room available in the FIFO. Lines 4 to 6 are repeated several times.

3. IP CORE INTEGRATION WITH ESPAM

In the previous sections, we presented our methodology for multiprocessor system design implemented in ESPAM. It allows automated generation of homogeneous multiprocessor platforms, that is, the processing components are only programmable processors. However, in many cases, a homogeneous system may not meet the performance requirements of an application. As an alternative, better performance can be achieved by using systems where different types of processing components execute different tasks. In general, a dedicated IP core delivers better performance than a processor which executes a software program for the same function. This motivated us to extend the ESPAM tool to support automatic generation of *heterogeneous* multiprocessor systems where both programmable processors and dedicated HW IP cores are used as processing components.

In this section, we present in detail our approach and techniques for automated IP core integration with ESPAM. It is based on a HW module generation that consists of a wrapper around a dedicated IP core. The basic idea in our approach is presented in Section 3.1. It is followed by a discussion on the type of the IPs supported by ESPAM, and the minimum interfaces these IPs have to provide in order to allow automated integration.. Then in Section 3.3, we give details about the internal structure of the HW module and the implementation of the wrapper. In Section 3.4, we explain how a HW module is automatically generated by ESPAM based on the KPN representation of the input application.

3.1. HW module—basic idea and structure

As we explained earlier, in the multiprocessor platforms we consider, the processors execute code implementing KPN processes, and communicate data between each other through FIFO channels mapped onto communication memories. Using communication controllers, the processors can be connected either point-to-point or via a communication component. We follow a similar approach to connect a dedicated IP core to other IPs or programmable processors in our platforms. We illustrate our approach with the example depicted in Figure 2. We map the KPN in Figure 2(b) onto the heterogeneous platform shown in Figure 2(c). Assume that process P1 is executed by processor *u*P2, P3 is executed by *u*P2, and the functionality of process P2 is implemented as a dedicated (predefined) IP core. Based on this mapping and the KPN topology, ESPAM automatically maps FIFO channels to communication memories (CMs) following the rule that a processing component only writes to its local CM. For example, process P1 is mapped onto processing component *u*P1 and P1 writes to FIFO channel CH1. Therefore, CH1 is mapped onto the local CM of *u*P1—see FIFO1 in Figure 2(c). In order to connect a dedicated HW IP core to other processing components, ESPAM generates a HW module (HM) that contains the IP core and a wrapper around it. Such an HM is then connected to the system using communication controllers (CCs) and communication memories (CMs), that is, an HM writes directly to its own local FIFOs and uses CC to read data from FIFOs located in CMs of other processors. This is illustrated in Figure 2(c)—see module HM that realizes process P2.

As explained in Section 2.3, our KPNs are derived automatically and the processes in our KPNs have always the same structure. It reflects the KPN operational semantics, that is, read-execute-write using blocking read/write synchronization mechanism. Therefore, a HW module realizing a process of our KPN has identical structure, shown in Figure 2(d), consisting of READ, EXECUTE, and WRITE blocks. A CONTROL block is added to capture the process behavior, for example, the number of process firings, and to synchronize the operation of the other three blocks.

The EXECUTE block of a HW module (HM) is actually a dedicated HW IP core to be integrated. It is not generated by ESPAM but it is taken from a library. The other blocks READ, WRITE, and CONTROL constitute the wrapper around the IP core. The wrapper is generated fully automatically by ESPAM based on the specification of a process to be implemented by the given HM. Each of the blocks in an HM has a particular structure which we illustrate with the example in Figure 3. Figure 3(a) shows the specification of process P2 generated by ESPAM if P2 would be executed on a programmable processor. We use this code to show the relation with the structure of each block in the HW modules generated by ESPAM, shown in Figure 3(b), if P2 is realized by dedicated HW.

In Figure 3(a), the read part of the code is responsible for getting data from proper FIFO channels at each firing of process P2. This is done by the code lines 5–8 which behave like a multiplexer, that is, the internal variable in_0 is initialized with data taken either from port IP1 or IP2. Therefore, the read part of P2 corresponds to the multiplexer MUX in the READ block of the HW module in Figure 3(b). Selecting the proper channel at each firing is determined by the if conditions at lines 5 and 7. These conditions are realized by the EVALUATION LOGIC READ component of block READ. The output of this component controls the MUX component. To evaluate the if conditions at each firing, the iterators of the for loops at lines 3 and 4 are used. Therefore, these for loops are implemented by HW counters in the HW module—see the COUNTERS component in Figure 3(b).

The write part in Figure 3(a) is similar to the read part. The only difference is that the write part is responsible for writing the result to proper channels at each firing of P2. This is done in code lines 10–13. This behavior is implemented by the demultiplexer DeMUX component in Figure 3(b). DeMUX is controlled by the EVALUATION LOGIC WRITE component which implements the if conditions at lines 10 and 12. Again, to implement the for loops, ESPAM uses the COUNTERS component. Although the counters correspond to the control part of process P2, ESPAM implements them in both the READ and WRITE blocks, that is, it duplicates the for loops implementation in the HW module. This allows the operation of blocks READ, EXECUTE, and WRITE to overlap, that is, they can operate in pipeline which leads to better performance of the HW module.

The execute part in Figure 3(a) represents the main computation in P2 encapsulated in the function call at code line 9. The behavior inside the function call is realized by the dedicated HW IP core depicted in Figure 3(b). As explained above, this IP core is not generated by ESPAM, but it is provided by a designer or it is a predefined third party IP core. In the HW module, the output of component MUX is connected to the input of the IP core and the output of the IP core to the input of component DeMUX. In our example, the IP core has one input and one output. In general, the number of inputs/outputs can be arbitrary. Therefore, every IP core input is connected to one MUX and every IP core output is connected to one DeMUX.

Notice that the loop bounds at lines 3-4 in Figure 3(a) are parameterized. The CONTROL block in Figure 3(b) allows the parameter values to be set/modified from outside the HW module at run time or to be fixed at design time. Another function of block CONTROL is to synchronize the operation of the other blocks and to make them to work in pipeline. Also, CONTROL implements the blocking read/write synchronization mechanism. Details are given in Section 3.3.

3.2. IP core types and interfaces

In this section, we describe the type of the IP cores that fit in our HW module idea and structure discussed above. Also, we define the minimum data and control interfaces these IPs have to provide in order to allow automated integration in MPSoC platforms designed with ESPAM.

(1) In our HW module, an IP core implements the main computation of a KPN process which in the initial specification is represented by a function call. Therefore, an IP core has to behave like a function call as well. This means that for each input data, read by the HW module, the IP core is *executed* and produces output data after an arbitrary delay.

(2) In order to guarantee seamless integration within the dataflow of our heterogeneous systems, an IP core must have unidirectional data interfaces at the input and the output that do not require random access to read and write data from/to memory. Good examples of such IP cores are the *multimedia cores* at http://www.cast-inc.com/cores/.

(3) To synchronize an IP core with the other blocks in our HW module, the IP has to provide "Enable/Valid" control interface. The "Enable" signal is a control input to the IP core and is driven by the CONTROL block in the HW module to enable the operation of the IP core when input data is read from input FIFO channels. If input data is not available, or there is no room to store the output of the IP core to output FIFO channels, then "Enable" is used to suspend the operation of the IP core. The "Valid" signal is a control output signal from the IP and is monitored by block CONTROL in order to ensure that only valid data is written to output FIFO channels connected to the HW module.

3.3. IP core wrapper implementation

As explained in Section 3.1, a HW module generated by our ESPAM tool contains a dedicated HW IP core and a wrapper around it. In this section, we give details about the structure and implementation of the IP Wrapper. We have already explained and shown that the structure of our HW

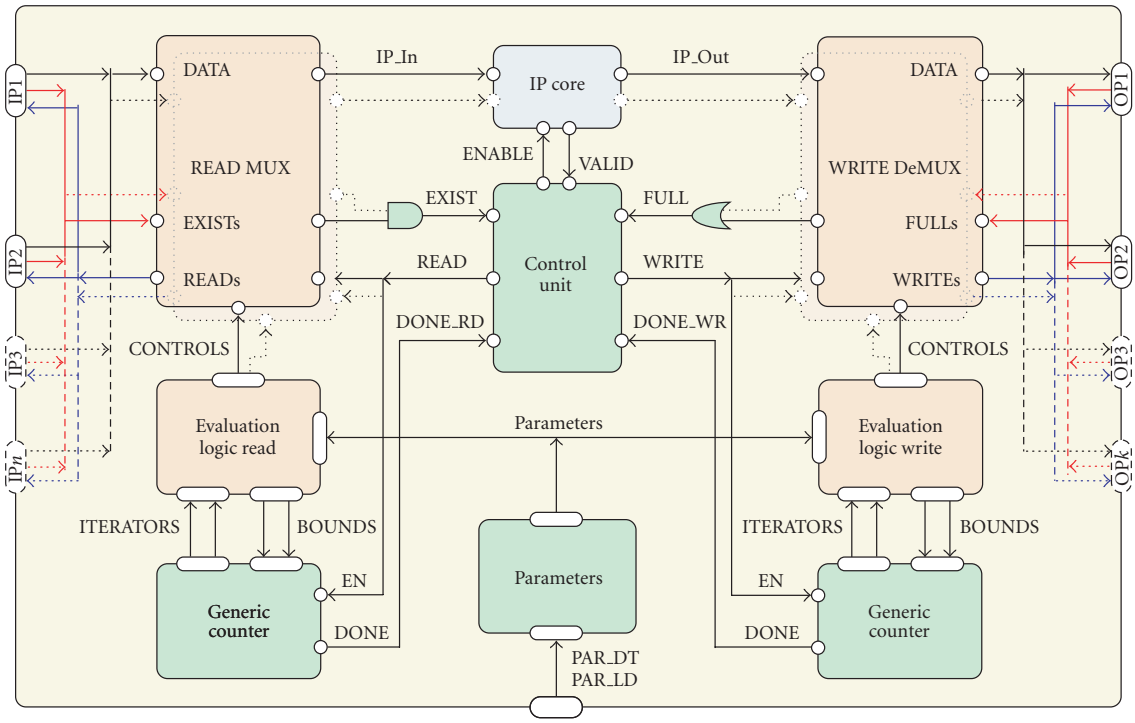


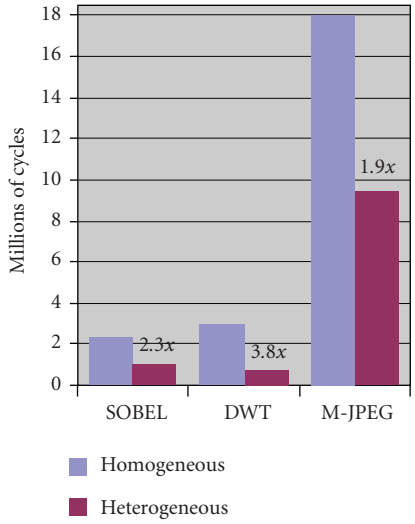
Figure 4: Detailed structure of the IP wrapper generated by ESPAM.

modules is the same regardless of the functional behavior they implement. This allows us to identify and implement the main building blocks discussed in Section 3.1 as a set of parameterized components with clearly defined interfaces, and by using these components to construct the IP Wrapper. Therefore, we have developed a HW module library of predefined parameterized components which ESPAM uses to generate HW modules. The generation is done by instantiating components from the library, connecting them, and setting their parameters according to the specification of processes in the input KPN. By making the IP wrapper (HW module, resp.) clearly structured and modularized, every component becomes more independent and loosely coupled. Therefore, we are able to develop and optimize each component separately which allows for efficient and effective optimization resulting in better performance of the generated systems. Below, we give more information about each component of the IP wrapper, its interface signals and its parameters.

The detailed structure of the IP wrapper is depicted in Figure 4. It contains several input and output data ports (IP1–IPn and OP1–OPk). The actual number is determined by the KPN topology. Each port implements a basic FIFO interface. It contains the minimum amount of signals that are present in any existing FIFO component. For the input ports, these are a DATA bus, EXIST, and READ signals. DATA is a bus used to read data from a FIFO connected to a port. EXIST is a signal indicating that a FIFO contains valid data to read and READ is a signal that controls the read enable of a FIFO. An output port implements a FIFO interface consisting of a DATA bus, FULL, and WRITE signals. DATA is a bus used to write data to an FIFO. FULL is a signal indicating when a FIFO is full and WRITE is a signal that controls the write enable of a FIFO. The IP wrapper has three global parameters defining its interfaces: number of input ports, number of output ports, and width of the data buses of the ports.

The data buses of the input interfaces are connected to the IP core inputs through multiplexers READ MUX. Recall that we use one READ MUX component for every input of an IP core. The READ MUX has two parameters: number of input ports (*#inputs*) and data width. Notice that each READ MUX port contains DATA bus, EXIST, and READ signals. Similarly, the outputs of the IP core are connected to the data buses of the output interfaces of the HW module using WRITE DeMUX components, one component per IP output. The parameters of component WRITE DeMUX are number of output ports (*#outputs*), each containing DATA bus, FULL, WRITE signals, and data width.

The CONTROL UNIT component, depicted in the center of Figure 4, synchronizes the operation of the IP wrapper components with the operation of the IP core. CONTROL UNIT does not have parameters. Therefore, its implementation and structure are exactly the same in any HW module generated by ESPAM. CONTROL UNIT generates READ and WRITE strobes to the FIFO channels and ENABLE signal to the IP core based on the EXIST and FULL signals coming from the FIFO channels. The DONE_RD and DONE_WR signals terminate the read/write actions. The implementation of CONTROL UNIT is very simple, that is, it implements only the three boolean equations described in Figure 6.



(a) Performance numbers

M-JPEG	Homogeneous		Heterogeneous	
VideoIn	11 K	4.1%	11 K	7.9%
DCT	135 K	50.8%	0.4 K	0.3%
Q	68 K	25.7%	68 K	50.3%
VLE	49 K	18.7%	49 K	39.5%
VideoOut	1.8 K	0.7%	1.8 K	2%

(b) M-JPEG tasks utilization

	SLICEs	FFs	LUTs
W1	221	190	371
W2	240	192	371
W3	208	147	361
W4	274	173	412
W5	269	173	390
W6	236	157	351

(c) HW resource utilization

Figure 5: Experimental performance and synthesis results.

READ = EXIST and not(FULL) and not(DONE_RD),

ENABLE = (not(DONE_RD) and READ), or
(DONE_RD and not(FULL) and not(DONE_WR))

WRITE = VALID and not(FULL) and not(DONE_WR).

Figure 6

The first equation shows the logic that enablesreading from FIFO channels. When data to be read is available, all the FIFOs where the result has to be stored are not full, and the read actions are not completed, signal READ is set to 1 and reading is initiated. The second equation is the logic that enables the operation of the IP core. It is enabled if the read actions are not completed and there is data to read. When the read actions are completed, the IP operation is enabled if the corresponding output FIFO channels are not full and the write actions are not completed. The third equation shows the logic that enables writing to FIFO channels. When the processed data is available at the output of the IP core indicated by the VALID signal, all the FIFOs where the data has to be stored are not full, and write actions are not completed, signal WRITE is set to 1, and writing is initiated.

As explained in Section 3.1, for loops are implemented as counters. Therefore, we have designed a parameterized GENERIC COUNTER component where only by setting parameters arbitrary nested for loops can be implemented. The parameters define the number of counters (*#counters*) and the *size* of each counter. GENERIC COUNTER has interface called BOUNDS that allows the lower and upper bounds of each counter to be set at run time. The GENERIC COUNTER components are controlled by enable signals (EN), that is, if a read or write action has been performed, the corresponding counter advances. When all the counters reach their upper bounds, this situation is indicated to CONTROL UNIT by signals DONE_RD and DONE_WR.

The parameters of the components discussed so far are so called *structural* parameters. They are set at design time and used to determine the components structure. However, processes in our KPNs may contain parameters that determine at run time the behavior of the processes. For example, the upper bounds of the for loops of process P2 in Figure 3(a) are parameterized by parameters N and M. We call such parameters *behavioral* parameters. These parameters are managed in our HW module by the PARAMETERS component. It provides an interface, PAR_DT, and PAR_LD, for loading behavioral parameter values from outside of the HW module at run time. PAR_DT is a data bus to transfer behavioral parameter values and PAR_LD is the write enable signal to store them in the PARAMETERS component. Like most of the components in our HW module, the PARAMETERS component itself has structural parameters that define the number of behavioral parameters (*#parameters*) and their default *values*.

The behavioral parameters in our KPNs are used to determine for-loop bounds and to evaluate if conditions that determine input/output ports selection as shown in Figure 3(a). In our IP wrapper, this is implemented in the two EVALUATION LOGIC components shown in Figure 4. The values of the behavioral parameters are propagated from the PARAMETERS component to these two components. Also, the values of the loop iterators implemented as counters in the GENERIC COUNTER components are propagated through the ITERATORS buses. In the two EVALUATION LOGIC components, the counter bounds are calculated and propagated back to the GENERIC COUNTER components at run time through the BOUNDS interface. Also, in the two EVALUATION LOGIC components the selection conditions are calculated to determine the values of signals CONTROLS.

These signals control components READ MUX and WRITE DeMUX that select from/to which FIFO channels data has to be read/written. The structural parameters of the EVALUATION LOGIC components are number of output control signals (*#control_signals*), number of counters (*#counters*), and number of behavioral parameters (*#parameters*). We would like to notice that the implementation of components EVALUATION LOGIC depends not only on the values of the structural parameters, but also on the specification of a KPN process realized by a HW module. Therefore, ESPAM synthesizes different EVALUATION LOGIC components for each HW module in our heterogeneous platforms. In contrast, the other components in any HW module are predefined and ESPAM just instantiates them from the HW module library and only assigns proper values to their structural parameters.

3.4. Automated generation of HW modules

The presented components of our HW module, their parameters and interfaces as well as the structure of the HW module allow the generation of the HW module to be automated in ESPAM. In this section, we explain how ESPAM automatically generates a HW module based on a KPN process specification. This is done in several steps, namely, components instantiation, parameters setting, and evaluation logic generation. For the sake of clarity, we illustrate these steps by an example of generating the HW module for process P2 in Figure 2(b). The process specification is given in Figure 3(a). The generated HW module is the module depicted in Figure 4 if the dashed components and connections are not considered.

First, ESPAM instantiates all the components of the IP wrapper and the IP core as follows. The IP wrapper requires two input (IP1, IP2) and two output (OP1, OP2) ports. This information is extracted from the KPN topology—see process P2 in Figure 2(b). The IP core that implements the main computation of process P2 in our example has one input and one output port. Therefore, ESPAM instantiates one READ MUX and one WRITE DeMUX components and connects them to the IP core and to the input/output ports of the HW module. Then ESPAM instantiates and connects the remaining components of the IP wrapper as shown in Figure 4. This is possible to be done automatically because the number of the remaining components and their connections do not change across different HW modules and it is predefined as a parameterized template in ESPAM. Second, ESPAM sets the values of the parameters of the instantiated components as follows:

READ MUX—*#inputs* = 2 because process P2 has two input ports;

WRITE DeMUX—*#outputs* = 2 because process P2 has two outputs;

PARAMETERS—*#parameters* = 2 because process P2 has two behavioral parameters N and M. The default values of N and M are set as well by extracting this information from the specification of process P2. This is possible because in our KPNs a behavioral parameter is defined by min, max, and default values;

```
BOUND_LOW_i <= 2;
BOUND_HIGH_i <= PAR_N;
BOUND_LOW_j <= 1;
BOUND_HIGH_j <= PAR_M + ITERATOR_i
```

Figure 7

```
CONTROLS(0) <= (−iterator_i + par_N − 1) >= 0; - - OP1,
CONTROLS(1) <= (iterator_i − par_N) = 0; - - OP2
```

Figure 8

GENERIC COUNTER—*#counters* = 2 because the body of process P2 contains two nested for-loops with iterators i and j. The parameter *size* of each counter is set according to the maximum value that each loop iterator can reach during operation. In our example, size = max(N) for the counter corresponding to loop iterator i. Similarly, size = max(M) + max(N) for the counter corresponding to loop iterator j;

CONTROL UNIT—no parameters are set because, as we explained in Section 3.3, this component is not parameterized;

EVALUATION LOGIC—*#control_signals* = 2 because of the two input and two output ports of the HW module, *#counters* = 2 because process P2 has two nested loops, and *#parameters* = 2 because P2 has two behavioral parameters N and M.

In our example, parameter *data_width* of all components is set to 32 bits.

Finally, as a last step of the automated generation of the HW module, ESPAM generates the implementations of the EVALUATION LOGIC components. The implementations contain two parts, that is, logic to calculate the lower and upper bounds of the counters and logic to calculate the values of the control signals propagated to the READ MUX and WRITE DeMUX components. According to the specification of process P2 at lines 3-4 in Figure 3(a), the counter bounds in both EVALUATION LOGIC components are implemented by logic synthesized from VHDL code generated by ESPAM as shown in Figure 7.

The read if conditions at code lines 5 and 7 and the write if conditions at code lines 10 and 12 in Figure 3(a) are also implemented by logic synthesized from VHDL code generated by ESPAM. In Figure 8, we show the code of the write if conditions in the EVALUATION LOGIC WRITE component.

4. EXPERIMENTS AND RESULTS

In this section, we present some of the results we have obtained by implementing and executing three applications, namely, a Sobel edge detection, a discrete wavelet transform

(DWT), and a motion JPEG (M-JPEG) encoder application, onto homogeneous and heterogeneous multiprocessor systems using our ESPAM tool and the design flow presented in Section 2.1. The main objective of this experiment is to show the effectiveness of our approach for HW IP core integration in terms of design time, achieved performance, and HW resource utilization of the generated HW modules. For prototyping purpose, we use an FPGA board with one Xilinx VirtexII-6000 device.

4.1. Design time

Following our design flow, we started with the three applications (Sobel, DWT, and M-JPEG) given as sequential C programs and automatically derived the *Application Specifications*, that is, KPNs using the PNgen tool in 5 minutes. Details about the derived KPNs are presented in [19]. For each application, we wrote the system-level *Platform* and *Mapping Specifications* by hand in XML format in 10 minutes. In this experiment, each of the three homogeneous platforms contains 5 *MicroBlaze* processors connected via crossbar network. Having all three input specifications for each application, our ESPAM tool generated and programmed a homogeneous multiprocessor system at RTL level, which was automatically imported to the Xilinx XPS tool for physical implementation onto our prototyping FPGA. The overall design and implementation time of each homogeneous system was about an hour.

We have performed similar actions as described above in order to generate three *heterogeneous* multiprocessor systems using our design flow. We had to modify only the initial system-level *Platform* and *Mapping Specifications* for each application in order to replace some of the *MicroBlaze* processors with dedicated HW IP cores. This took us less than 5 minutes. For the Sobel application, we used 3 *MicroBlaze* processors and 2 dedicated IP cores. The IP cores estimate the first derivative of an image intensity function. For the DWT application, we used 1 *MicroBlaze* processor and 4 dedicated IP cores. The IP cores are 2 low and 2 high-pass filters. For the M-JPEG application, we used 4 *MicroBlaze* processors and 1 discrete cosine transform (DCT) IP core. Again, the overall design and implementation time of each heterogeneous system was about an hour.

As explained above, in the heterogeneous systems we used several dedicated HW IP cores. They were written in synthesizable VHDL. For the Sobel and DWT applications, the IP cores have a simple structure, that is, they implement convolution-based operations. These IP cores have been developed by 2 master students and added to the ESPAM library in about one working day. For the M-JPEG application, we used an IP core that performs a discrete cosine transform (DCT) operation. We have downloaded this IP core from the Xilinx website at http://www.xilinx.com/bvdocs/appnotes/xapp_610.zip. In order to add this IP core to the ESPAM library, we had to make small modification related to the control (Enable/Valid) interface discussed in Section 3.2. The DCT IP core provided by Xilinx has "Valid" signal but it does not have "Enable" signal. We added this signal to the IP core and the IP core to the library within 30 minutes.

4.2. Performance results

The performance numbers we have obtained for the implemented multiprocessor systems are shown in Figure 5(a). For each multiprocessor system, we measured the exact number of clock cycles needed to process an image of size 128×128 pixels. As one may expect, the numbers in the figure show that the heterogeneous systems achieve better performance. This is because the dedicated HW IP cores, we use, work more efficiently than the *MicroBlaze* processors for the same functionality. What is more important to discuss here is the achieved speed-up, depicted in Figure 5(a) above the bars of the heterogeneous systems, in order to show the efficiency of our approach for HW module generation and IP core integration. Consider the performance results of the M-JPEG systems. The M-JPEG application consists of 5 tasks, namely, VideoIn, DCT, quantization (Q), variable length encoding (VLE), and video out. The left part of column HOMOGENEOUS in Figure 5(b) shows how many thousands of clock cycles it takes for a *MicroBlaze* processor to execute each task by processing one data token—an image block of size 8×8 pixels. The numbers in the next column show the same information in percentage of the overall processing time utilized by each task. It can be seen that the DCT is the bottleneck of the system taking more than 50% of the whole processing time for one block and consequently, for the whole image. These 50% mean that if the DCT is substituted with more efficient implementation, theoretically the overall performance of the system can be increased at most 2 times. The column HETEROGENEOUS in Figure 5(b) shows the clock cycles and the percentage of each task performed by the heterogeneous M-JPEG system where the DCT is implemented by a very fast dedicated HW IP core and integrated using our HW module generation approach. In this system, the DCT takes only 0.3% of the whole processing time. In this case, Figure 5(a) shows that the overall speed-up compared to the homogeneous system is 1.9x which is close to the theoretical maximum 2x for the heterogeneous system where only the DCT is a dedicated IP core. This clearly shows the efficiency of our approach for IP core integration by generating HW modules.

4.3. Synthesis results

Recall that a HW module generated by our tool ESPAM consists of an IP core and a wrapper around it where the IP core is given and only the wrapper is generated by ESPAM. Therefore, we present only the HW resource utilization of our generated wrappers in order to show how efficient our wrappers are in terms of utilized HW. In Figure 5(c), we present the resource utilization of the IP wrappers of six IP cores that we used in our experiments. Each row W1–W6 in Figure 5(c) corresponds to an IP wrapper. The utilized FPGA resources are grouped into slices that contain flip-flops and 4-input look up tables—see columns 2, 3, and 4, respectively. The numbers show low HW resource utilization which on average is 241 slices. Moreover, the number of resources utilized by a wrapper does not depend on the size of the IP core it integrates, that is, a larger IP core does not require a

larger wrapper. For example, wrapper W3 of the DCT core utilizes only 208 slices, whereas the DCT IP core itself utilizes 1369 slices. Wrapper W2 of the IP core that estimates the first derivative in Sobel utilizes 240 slices, whereas the IP core itself utilizes 424 slices.

In general, the HW complexity of our wrappers is determined mainly by the number of MUX and DeMUX components, the number of counters implementing for loops of a KPN process, and the number of behavioral parameters of a KPN process. The three applications we used in our experiment process images. We specified the applications with two nested for loops that iterate through an image and we used two behavioral parameters as loop bounds, that is, image, width, and height. Since the number of for loops and behavioral parameters is the same for all wrappers in our experiment, the difference in the resource utilization of our wrappers is caused by the different input/output ports of the wrappers and the IP cores they integrate.

5. CONCLUSIONS

In this paper, we presented our method and techniques implemented in ESPAM for automated integration of dedicated hardwired IP cores into heterogeneous multiprocessor systems. The integration is based on a HW module generation that consists of predefined dedicated IP core and a wrapper around it. To allow automated IP core integration, our approach requires these IP cores to provide simple data and control interfaces. The proposed method for IP core integration was applied on several real-life applications. The reported results show that the integration is efficient in terms of performance and HW resource utilization.

ACKNOWLEDGMENT

This work was supported by the Dutch Technology Foundation (PROGRESS/STW) under the Artemisia project (LES.6389).

REFERENCES

[1] IBM PowerPC Wite Paper, http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC 405 Embedded Cores.

[2] The Xilinx's Microblaze Processor, http://www.xilinx.com/products/design resources/proc central/microblaze arc.htm.

[3] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: the next design discontinuity," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '02)*, pp. 84–90, Freiburg, Germany, September 2002.

[4] Tensilica, "Xtensa configurable processors," http://www.tensilica.com/products/xtensa-overview.htm.

[5] Intel Corp., "Intel IXP1200 Network Processor, Product Datasheet," December 2001.

[6] NXP, http://www.nxp.com/.

[7] M. Gries and K. Keutzer, *Building ASIPs: The Mescal Methodology*, Springer, New York, NY, USA, 2005.

[8] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon)*, Morgan Kaufmann, San Francisco, Calif, USA, 2006.

[9] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor systen design, programming, and implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 542–555, 2008.

[10] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress on Information Processing*, pp. 471–475, North-Holland, Stockholm, Sweden, August 1974.

[11] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: the Compaan/Laura approach," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 1, pp. 340–345, Paris, France, February 2004.

[12] E. A. de Kock, G. Essink, W. J. M. Smits, et al., "YAPI: application modeling for signal processing systems," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 402–405, Los Angeles, Calif, USA, June 2000.

[13] A. Nieuwland, J. Kang, O. P. Gangwal, et al., *C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems*, Kluwer Academic Publishers, Norwell, Mass, USA, 2002.

[14] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–112, 2006.

[15] E. A. de Kock, "Multiprocessor mapping of process networks: a JPEG decoding case study," in *Proceedings of the 15th International Symposium on System Synthesis (ISSS '02)*, pp. 68–73, Kyoto, Japan, October 2002.

[16] K. Goossens, J. Dielissen, J. van Meerbergen, et al., "Guaranteeing the quality of services in networks on chip," in *Networks on Chip*, pp. 61–82, Kluwer Academic Publishers, Hingham, Mass, USA, 2003.

[17] B. K. Dwivedi, A. Kumar, and M. Balakrishnan, "Automatic synthesis of system on chip multiprocessor architectures for process networks," in *Proceedings of the 2nd International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '04)*, pp. 60–65, Stockholm, Sweden, September 2004.

[18] G. Martin, "Overview of the MPSoC design challenge," in *Proceedings of the 43rd Design Automation Conference (DAC '06)*, pp. 274–279, San Francisco, Calif, USA, July 2006.

[19] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: a tool for improved derivation of process networks," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 75947, 13 pages, 2007.

[20] "Espam and PNgen download link," http://daedalus.liacs.nl/Site/Download.html.

[21] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures," in *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES '00)*, pp. 13–17, San Diego, Calif, USA, May 2000.

[22] T. Stefanov and E. Deprettere, "Deriving process networks from weakly dynamic applications in system-level design," in *Proceedings of the 1st International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 90–96, Newport Beach, Calif, USA, October 2003.

[23] A. Turjan, B. Kienhuis, and E. Deprettere, "Translating affine nested-loop programs to process networks," in *Proceedings of*

the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '04), pp. 220–229, Washington, DC, USA, September 2004.

[24] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "Laura: leiden architecture research and exploration tool," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL '03)*, pp. 911–920, Lisbon, Portugal, September 2003.

[25] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*, vol. 1132 of *Lecture Notes in Computer Science*, pp. 79–103, Springer, London, UK, 1996.

[26] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, "The liberty simulation environment: a deliberate approach to high-level system modeling," *ACM Transactions on Computer Systems*, vol. 24, no. 3, pp. 211–249, 2006.

[27] F. Doucet, S. Shukla, M. Otsuka, and R. Gupta, "BALBOA: a component-based design environment for system models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 12, pp. 1597–1612, 2003.

[28] D. A. Mathaikutty and S. K. Shukla, "MCF: a metamodeling based visual component composition framework," in *Advances in Design and Specification Languages for Embedded Systems*, pp. 319–337, Springer, New York, NY, USA, 2007.

[29] E. A. Lee, J. Liu, X. Liu, et al., "Ptolemy II: heterogeneous concurrent modeling and design in java," Tech. Rep. UCB/ERL M99/40, University of California, Berkeley, Calif, USA, 1999.

[30] H. D. Patel, S. K. Shukla, and R. A. Bergamaschi, "Heterogeneous behavioral hierarchy extensions for SystemC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 4, pp. 765–780, 2007.

[31] Celoxica, http://www.celoxica.com/.

[32] J. Zhu, R. Doemer, and D. D. Gajski, "Syntax and semantics of the specC language," in *Proceedings of the 7th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI '97)*, p. 8, Osaka, Japan, December 1997.

[33] C. Haubelt, J. Falk, J. Keinert, et al., "A systemC-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 47580, 22 pages, 2007.

[34] J. Falk, C. Haubelt, and J. Teich, "Efficient representation and simulation of model-based designs in systemC," in *Proceedings of the International Forum on Specification & Design Languages (FDL '06)*, pp. 129–134, Darmstadt, Germany, September 2006.

[35] A. A. Jerraya, A. Bouchhima, and F. Pétrot, "Programming models and HW-SW interfaces abstraction for multiprocessor SoC," in *Proceedings of the 43rd Design Automation Conference (DAC '06)*, pp. 280–285, San Francisco, Calif, USA, July 2006.

[36] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 518–523, Las Vegas, Nev, USA, June 2001.

[37] P. G. Paulin, C. Pilkington, M. Langevin, et al., "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 7, pp. 667–679, 2006.

[38] VSIA, http://www.vsi.org/.

[39] OCP, http://www.ocpip.org/.

[40] SPIRIT, www.spiritconsortium.org/.

[41] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink, "Design and programming of embedded multiprocessors: an interface-centric approach," in *Proceedings of the 2nd International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '04)*, pp. 206–217, Stockholm, Sweden, September 2004.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 685128, 13 pages
doi:10.1155/2008/685128

Research Article

A Case Study: Quantitative Evaluation of C-Based High-Level Synthesis Systems

Omar Hammami,[1] Zhoukun Wang,[1] Virginie Fresse,[2] and Dominique Houzet[3]

[1] *EECS Department, National Higher School of Advanced Techniques (ENSTA), 32 Boulevard Victor, 75739 Paris, France*
[2] *Hubert Curien Laboratoire (UMR CNRS 5516), Université Jean Monnet, 18 Rue Benoit Lauras, 42000 Saint Etienne, France*
[3] *Laboratoire Grenoble Images Paroles Signal Automatique (GIPSA Laboratoire) (UMR CNRS 5216), Institut National Polytechnique de Grenoble (INPG), 46 Avenue Félix Viallet, 38031 Grenoble, France*

Correspondence should be addressed to Omar Hammami, profhammami@yahoo.com

Received 5 October 2007; Revised 25 February 2008; Accepted 14 May 2008

Recommended by Sandeep Shukla

Embedded systems require a high-level abstraction modeling to tackle their complexity and improve design productivity. C-based design techniques and methodologies have been proposed for this purpose. Various IPs with diverse complexities and functionalities can be selected to build a system. However, area and energy issues are paramount in this selection process. Implementation hints should therefore be available at the highest possible level of abstraction. Some C-based tools have been proposed in this order through either true synthesis or estimation. In this paper, we conduct through a case study an evaluation of C-based design of embedded systems and point out the impact of behavioral synthesis on embedded systems multiobjective high-level partitioning.

1. INTRODUCTION

Embedded systems are increasingly complex to design, validate, and implement [1]. System composition of embedded processors and hardware accelerators leads to the HW/SW codesign methodologies. Traditional codesign methodologies require that hardware and software are specified and designed independently. Hardware and software validations are thus done separately without interface validations. The partitioning is therefore decided in advance and as changes to the partition can necessitate extensive redesign elsewhere in the system, this decision is adhered as much as possible. It can lead to suboptimal designs as partitioning techniques often rely on the designer's experience. This lack of a unified hardware-software representation can lead to difficulties in verification of the entire system, and hence to incompatibilities across the hardware/software boundary. Using a single language for both simplifies the migration task and ensures an entire system verification. Important uses of a unified design language in addition to synthesis are validation and algorithm exploration (including an efficient partitioning). C-like languages are much more compelling for these tasks, and one, in particular SystemC [2], is now widely used for system-level design [3–5], as are many ad hoc variants. Rapid C synthesis allows also fast emulation techniques of hardware and software used for architecture exploration. C-based synthesis fundamentals emerged from these HW/SW codesign methodologies with the objective of reducing the design productivity challenge [1]. However, although raising the system design level of abstraction contributes to reduce the design complexity, reliable and predictable silicon implementation remains mandatory which supports high-level design handoff [1]. The question is then on the impact of C-based behavioral synthesis on C-based specification and modeling framework for embedded systems.

This work evaluates through a case study the performance and efficiency of several high-level description languages (SystemC, Handel-C) for FPGA-based embedded systems. The rest of the paper is organized as follows: Section 2 presents the related work while Section 3 reviews main issues regarding C-based synthesis system design methodology. Design case studies along with performance evaluation studies and results are fully described in Section 4. Finally, Section 5 presents the conclusions.

2. RELATED WORK

Intensive research has been conducted on C-based behavioral synthesis. With the evolution of system-level design languages, the interest in efficient hardware synthesis based on behavioral description of a hardware module has also been visible. For a system designer, the behavioral synthesis is very attractive for hardware modeling as it leads to significant productivity improvements. Important work has been done in academia on behavioral synthesis with C/SystemC [6–14]. For example, the work in [6] has presented a synthesis environment and the corresponding synthesis methodology. It is based on traditional compiler generation techniques, which incorporate SystemC, VHDL, and Verilog to transform existing algorithmic software models into hardware system implementations. In [12], the authors address the problem of deriving exact finite state machines (FSMs) from SystemC descriptions which is the first part of behavioral synthesis methodologies. In an effort to extend synthesis to object-oriented constructs of C++ language, [10] presents an approach to object-oriented hardware design and synthesis based on SystemC. Behavior synthesis problem is multiobjective in nature where synthesis tools allow the hardware designers to customize the behavioral synthesis process through various options which constitute a huge design space. In this situation, solution exists of a set of optimized results represented in the form of Pareto curves and Pareto surfaces. In [14], a methodology that allows the designers to generate and analyze the best synthesis results based on area, performance, and power consumption estimation through an automatic exploration of synthesis results is presented. ROCCC [15] and Spark [11] are C-to-VHDL high-level synthesis academic frameworks.

Some tools for behavioral SystemC/C synthesis are available in the market. Synopsys SystemC compiler [16] was perhaps the first commercial effort to synthesize behavioral code written in ESL languages. Celoxica's agility [17] and Forte Design [18] can synthesize a SystemC behavioral description of hardware modules. They also give the area estimation requirements for various ASIC and FPGA-based architectures. Orinoco Dale [19], ImpulseC [20], and CatapultC [21] estimate the area and the energy for a C description of an application.

C-based behavioral synthesis can be used at system-level design in order to guide system partitioning. Reference [4] presents a framework for the generation of embedded hardware/software from SystemC. In [5], area and energy are estimated for various IP components in the system before actually modeling the system in TLM. The area and energy estimation information is fed into the TLM model of the system where we partition the system by automatically exploring the system design space based on the given information. This design flow uses the TLM modeling for system design space exploration. It includes area and energy estimation information during the partitioning process. This work represents a good layout foundation for addressing the impact of implementation on embedded systems.To the best of our knowledge, our study is the first attempt to the contribution of benchmarking these tools for performance comparison purposes and to analyze the interaction with place and route tools options.

3. C-BASED SYNTHESIS

3.1. C-language fundamentals

Some C-language characteristics are troubling when synthesizing hardware from C. Edwards listed in [22] the key challenges of a successful hardware specification language: concurrency model, types, timing specification, memory and communication patterns, hints, and constraints. The C-language has no support for parallelism. Either the synthesis tool is responsible for finding it or the designer must modify the algorithm by inserting explicit parallelism. C-hardware language designers adopted different parallelism strategies. Communication patterns depend on the parallel programing model provided by the C-hardware languages. These communication patterns do not exist in C-language. The C-language is also mute on the subject of time. It guarantees causality among most sequences of statements but says nothing about the amount of time it takes to execute each. It is essential to find reasonable techniques for specifying hardware needs and mechanisms for specifying and achieving timing constraints. Data types are another major difference between hardware and software languages. The most fundamental type in hardware is a single bit traveling through a memoryless wire. Variable width integer is natural in hardware yet C does not support variable width. C's memory model is a large undifferentiated array of bytes, yet many small varied memories are most effective in hardware. All these characteristics must be considered when designing C-like hardware languages. All characteristics related to the considered tools are analyzed in the rest of the paper.

3.2. HLS approaches and tools

Various C-based hardware description languages have been proposed over the past decade. These tools use different approaches for timing, parallelism, data types, and communication modeling. These approaches can be either automatic or manual.

(i) Concurrency model. Coarse grain and fine grain parallelism are available for most of the approaches. The communication between tasks is the coarse grain parallelism. The fine grain parallelism is the operator parallelism and pipeline. This parallelism is either explicit or implicit. Constructs dedicated to the parallelism are added to the C-language for the explicit parallelism programing. Additional constructs are not required for the implicit parallelism. The level of parallelism can be handled with constraints and compile options.

(ii) Types. All approaches ensure hardware bit-true data type manipulation and declaration with additional data types. The size of the data can be precisely controlled for each operating stage.

Table 1: Approaches used for C-based hardware description languages.

Features	Approaches	Language	Tools
Concurrency model	Implicit	C	Spark, Impulse CoDeveloper, CatapultC, ROCCC,
	Explicit	Handel-C, SystemC	DK Design Suite, Forte Cynthesizer, and agility
Types	Explicit	All languages	All tools
Timing specification	Implicit	C	Spark, Impulse CoDeveloper, CapatultC, ROCCC,
	Explicit	Handel-C, SystemC	DK Design Suite, Forte Cynthesizer, and agility
Memory	Implicit	C	Impulse CoDeveloper
	Explicit	Handel-C, C	DK Design Suite
Communication patterns	Implicit	C	Spark
	Explicit	Handel-C	DK Design Suite, Impulse CoDeveloper, and agility

Table 2: Case study selected C-based environments.

Tool	Language	Company
Impulse CoDeveloper	ImpulseC	Impulse Accelerated Technologies
DK suite	Handel-C	Celoxica
Agility SystemC compiler	SystemC	Celoxica

(iii) Timing specification. All approaches take the latency and the throughput into account. The approaches are either explicit or implicit. Each clock cycle is precisely described for the explicit approach. The timing constraint only concerns the clock period. Tools with an implicit timing approach generate the scheduling and the parallelism of operators to meet the latency and the throughput timing constraints.

(iv) Memory. Internal memories are either RAM or registers. They can be either implicitly selected or explicitly specified.

(v) Communication patterns. Predefined communication and memory protocols are used for the implicit approach. For the explicit approach, the protocols are described in details in the code with dedicated instructions.

The chosen approach can lead to substantial code modifications.

Each tool uses specific approaches for concurrency model, types, timing specification, and memory and communication pattern. Several tools cannot be studied in this paper but they can be classified in the following table (see Table 1).

Several tools can be evaluated with the presented approaches. Three selected tools given in Table 2 are selected on the basis of our own design experience and the used approaches. It is indeed of importance that a significant amount of design experience is needed in order to compare the design approaches with the design. Each tool has different approaches for concurrency, data types, timing specifications, memory and communication.

3.2.1. ImpulseC

Impulse CoDeveloper is an ANSI C synthesizer [20] based on the ImpulseC language with function libraries supporting embedded processors and abstract software/hardware communication methods including streams, signals, and shared memories. This allows software programmers to make use of available hardware resources for coprocessing without writing low-level hardware descriptions. Software programmers can create a complete embedded system that takes advantage of the high-level features provided by an operating system while allowing the C programing of custom hardware accelerators. The ImpulseC tools automate the creation of required hardware-to-software interfaces, using available on-chip bus interconnections.

(i) *Concurrency model*. The main concurrency feature is pipelining. As pipelining is only available in inner loops, loop unrolling becomes the solution to obtain large pipelines. The parallelism is automatically extracted. Explicit multiprocess is also possible to manually describe the parallelism.

(ii) *Types*. ANSI C-type operators are available like "int" and float as well as hardware types like "int2," "int4," and "int8." The float to fixed point translation is also available.

(iii) *Timing specification*. The only way to control the pipeline timings is through a constraint on the size of each stage of the pipeline. The number of stages of the pipeline and thus the throughput/latency are tightly controlled.

(iv) *Memory*. All arrays are stored either in RAM or in a set of registers according to a compilation option.

(v) *Communication patterns*. Streams (FIFO) with different formats are available as well as signals and shared memories interface.

3.2.2. DK design suite tool

DK Design Suite is a complete Electronic System Level (ESL) environment supporting the Handel-C language [23]. It provides the user with a complete flow: from specification to implementation such as architecture-optimized EDIF

Netlist for FPGA's RTL Verilog or VHDL used for alternative synthesis flows and other hardware targets including ASIC designs.

Celoxica's Handel-C is a language for digital logic design that has many similarities to ANSI-C. Handel-C is a variant that extends the language with constructs for parallel statements and OCCAM-like rendez-vous communication.

Handel-C language and the IDE tool introduced by Celoxica provide both simulation and synthesis capabilities.

(i) *Concurrency model*. The application is written with sequential programs in Handel-C. Programs written in Handel-C are implicitly sequential: writing one command after another indicates that those instructions should be executed in that exact order. Parallel constructs are possible with the *par* keyword to gain maximum benefit in performance from the target hardware.

(ii) *Types*. This language adopts the manual customization of numerous representations. Handel-C types are not limited to specific widths. Any defined Handel-C variable can be specified with the minimum width required to minimize hardware usage.

(iii) *Timing specification*. DK Design Suite includes a cycle accurate multithread symbolic debugger. Handel-C timing model is uniquely simple: any assignments or delay take one clock cycle.

(iv) *Memory*. Each data storage is explicitly specified in a RAM or a set of registers by the programmer.

(v) *Communication patterns*. Any external specification with any type of communication protocol can be described. Handel-C allows you to target components such as memory, ports, buses, and wires.

3.2.3. Agility compiler

The agility compiler from Celoxica is described below [17]. The agility compiler provides a behavioral design and synthesis for SystemC. It is a single solution for FPGA design and ASIC/SoC prototyping. Early SystemC models can be quickly realized in working silicon yielding accurate design metrics and RTL for physical design.

(i) *Concurrency model*. Explicit multiprocess is possible to manually manage parallelism. Each stage of the pipeline can be manually described with the use of wait() instructions in an RTL-like style.

(ii) *Types*. ANSI C types and operators such as "int" and "char" can be used. Agility compiler also accepts hardware types such as sc_uint⟨8⟩, sc_int⟨20⟩, and fixed point sc_fix⟨⟩.

(iii) *Timing specification*. The SystemC timing model is uniquely simple: all assignments between two wait() take one clock cycle. This modeling style is similar to the VHDL "wait until rising_edge(clk)" style.

(iv) *Memory*. SystemC provides supports for interfacing to on-chip RAMs and ROMs using dedicated array keywords. If not used, arrays are stored in a set of registers.

(v) *Communication patterns*. Any external specification with any type of communication protocol can be described.

4. DESIGN CASE STUDIES

In order to evaluate the synthesis efficiency of the previously described tools, the use of commonly accepted benchmarks for C-based synthesis would have been useful. However, so far no benchmarks have been released from the OSCI synthesis working group which defined the synthesizable subset of neither SystemC nor by any other body. Therefore, we decided to compose our own case studies which are basic and simple functions to ensure reproducibility.

4.1. Designs examples

Evaluation consists in studying the efficiency of the synthesis from C-based hardware descriptions. Common benchmarks are used for the evaluation of the previously described tools.

Our own case studies consist in a set of short and simple functions to allow reproducibility. The selected cases are two 3×3 image filters [24], the FFT, and an octree traversal algorithm (ray casting in projective geometry RCPG) [25].

4.1.1. Linear filter

A linear filter and a nonlinear filter are chosen. The two filtering benchmarks are based on a $3*3$ window core processing. The linear filter is the mean filter [24]. The mean filter is the simplest type of low-pass filter. The mean or average filter is used to soften an image by averaging surrounding pixel values in a 3×3 window. This filter is often used to smooth images prior to processing. It can be used to reduce pixel flicker due to overhead fluorescent lights.

4.1.2. Median filter

The second filter is the median filter based on the bubble sort of the $3*3$ neighboring pixels [24]. The median filter is a nonlinear digital filter which is able to preserve sharp signal changes and is very effective in removing impulse noise. This filter is widely used in digital signal and image/video processing applications. For the median filter, pixels are first sorted based on intensity. The center pixel would be the middle value of the sorted list of pixels. We present an example of ImpulseC code for the mean filter in Figure 1.

The filters are implemented by sliding a window of odd size (a $3*3$ window) over an $N*M$ image. At each window position, the sampled values are sorted and the resulting value of the sample replaces the sample in the center of the window. Three 32-bit streaming inputs provide four pixels for each line for each clock cycle. The size of the internal storage is $6*3$ pixels. Three internal storage solutions are implemented and evaluated. The first one is a sequential one with RAM as internal storage. The second one is a parallel/pipeline solution with RAM as

```
void FIL_MEAN(unsigned char image[][n],
unsigned char vimage[][vn])
{short x, y;
unsigned int sum;
   for (x = 1; x < vn − 1; x++)
      for (y = 1; y < vn − 1; y++)
      {
       sum = vimage[x][y];
       sum += vimage[x−1][y];
       sum += vimage[x + 1][y];
       sum += vimage[x − 1][y − 1];
       sum += vimage[x][y − 1];
       sum += vimage[x + 1][y − 1];
       sum += vimage[x − 1][y + 1];
       sum += vimage[x][y + 1];
       sum += vimage[x + 1][y + 1];
       image[x − 1][y − 1]= sum/9;
      }
```

(a) OriginalC code

```
while (co_stream_read(r0, & p0, sizeof(int32))
== co_err_none){
#pragma CO PIPELINE
#pragma CO set stageDelay 32
#pragma CO nonrecursive image0
#pragma CO nonrecursive image1
#pragma CO nonrecursive image2
co_stream_read (r1, & p1, sizeof(int32));
co_stream_read (r2, & p2, sizeof(int32));
for (k = 2; k < 6; k++){
#pragma CO UNROLL
image0[k]= p0 & 255; p0 = p0 ≫ 8; image1[k]= p1 & 255;
p1 = p1 ≫ 8;
image2[k]= p2 & 255;
p2 = p2 ≫ 8; }

res = 0;

for (k = 1; k < 5; k++){
#pragma CO UNROLL
res =((image0[k − 1]+ image0[k]+ image0[k + 1]
+ image1[k − 1]+ image1[k]+ image1[k + 1]
+ image2[k − 1]+ image2[k]+ image2[k + 1])≫ 3)
+(res ≪ 8);
}

for (k = 0; k < 2; k++){
#pragma CO UNROLL
temp = image0[k + 4];
image0[k]= temp;
temp = image1[k + 4];
image1[k]= temp;
temp = image2[k + 4];
image2[k]= temp; }
co_stream_write (output_stream, & res, sizeof(int32));
```

(b) ImpulseC code

Figure 1: The C and ImpulseC codes for a 3∗3 mean filter.

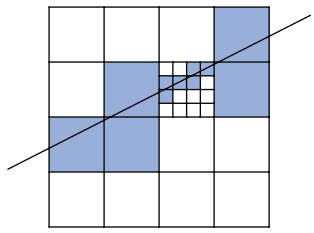


Figure 2: 2D recursive octree grid traversal principle.

internal storage. Three separate RAMs are used to allow parallelism between the three inputs. The third solution is a parallel/pipeline solution that uses registers as internal storage. Figure 2 represents the structure of external signals for both filters. Three streaming inputs are used to provide 4 pixels. Each input corresponds to one line of the image.

4.1.3. FFT

The third benchmark is the radix-4 FFT on 256 complex values (16-bit).

4.1.4. Ray casting in projective geometry (RCPG)

The ray casting in projective geometry (RCPG) is an iterative octree traversal algorithm (see Figure 2) [25]. In a regular grid, from a current cell crossed by the ray, the next cell is defined by a minimization of a cost function which is iteratively updated. At each step, the ray is propagated in 3D along the directions x, y, and z. The direction depends on the face where the ray and the current cell intersect (see Figure 2). The parameters of the intersection between the ray and each face are progressively stored and updated. The data structure is an octree structure: a cell can contain a data pointer to a higher resolution grid. Thus at each cell, the algorithm continues to the next one or descents in the subgrid. When it reaches a subgrid border, it mounts to the upper level of the grid.

For this design case, the sequential and pipeline algorithms are described.

The chosen cases are described in Table 3.

4.2. Target platform

The previously described case studies are intended to be synthesized, placed, and routed on a target technology in

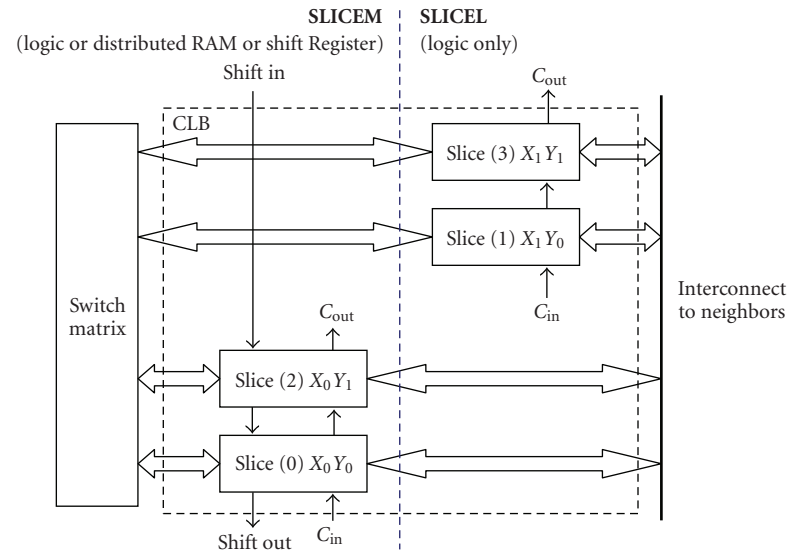


Figure 3: Virtex-4 slice structure. *Source: Xilinx Virtex-4 user guideUG070 (version 2.3)*.

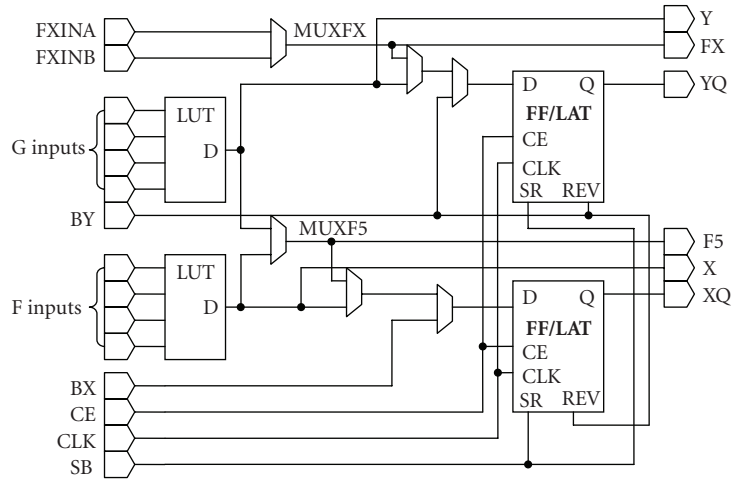


Figure 4: Virtex-4 slice L structure. *Source: Xilinx Virtex-4 user guideUG070 (version 2.3)*.

Table 3: Core case studies.

Benchmark	Description	No IP
Linear digital filter (mean)	Pipeline using registers	IP1
	Pipeline using memory	IP2
	Sequential	IP3
Nonlinear digital filter (median)	Pipeline using registers	IP4
	Pipeline using memory	IP5
	Sequential	IP6
FFT	Sequential	IP7
	Pipeline	IP8
RCPG	Sequential	IP9
	Pipeline	IP10

order to evaluate how the selected C-based design environment outputs as inputs to a same synthesis, place, and route tool will be processed. The metrics to be considered will be performance and area.

4.2.1. Target technology

We selected the FPGA Xilinx Virtex-4 technology [26] as the target technology in this case study. The main reason for the choice of an FPGA technology was to allow a quick implementation and verification of all the IPs through actual execution on chip. The Virtex-4 technology based on a 90 nm copper CMOS process has a fixed number of hardcores resources such as DSP, embedded RAM, and fixed interconnections. The Xilinx Virtex-4 can be described as a matrix of CLB each of them being composed of several slices (see Figure 3). The Xilinx Virtex-4 proposes memory-oriented slices *SLICEM* and logic-oriented slices *SLICEL* (see Figure 4). Also the embedded memory BRAM is a dual port 18 kb memory array. Hard cores in the FX family include embedded processors—IBM PowerPC—and 18×18 two's complement signed multipliers (DSP blocs).

The FPGA structure represents an additional challenge for C- and SystemC- based synthesis tools due to the

higher granularity and heterogeneity of FPGA compared to ASIC. The variety of FPGA resources makes the resource selection more difficult for the compiler tools to synthesize high-level C constructs. Several similar resources can be good candidates for one C-construct. The compiler tool has to select the most appropriate resource among all these candidate resources.

4.2.2. Tools and options combinations

Physical synthesis has been conducted using Xilinx tools ISE XST [27]. We conducted an intensive and wide automatic exploration of physical synthesis options (synthesis, place, and route). These range from area to speed oriented optimizations with in addition an exploration of the device density factor. All configurations have been executed on an FPGA board. This physical level design space exploration comes as a complement to high-level optimization techniques used by C-based synthesis such as for example, speculative execution (pipeline). The mutual effects—potentially inhibitory—of C-based synthesis followed by a VHDL physical synthesis are unspecified in any of the tool documentations. Different combinations of synthesis and place/route options on the different cases are explored in order to evaluate the possible interactions. The options used for VHDL synthesis and place and route are the global optimization options for area and speed with the level of optimization:

(1) XST VHDL synthesis option: *-opt_mode* speed or area,

(2) mapping option: *-cm* balanced or speed or area,

(3) place and route option: *-ol* std or med or high or *-ol* high -xe *n*.

This results in 24 combinations of synthesis, place and route options for each IP.

4.3. Timing results

Timing results are presented in Figure 5. The timing results are obtained for each IP with the use of the previously presented tools. The timing metrics are the clock frequency, the latency, and the number of cycles per result.

The variability of results between the tools comes from different reasons. Firstly, the RAM implementation is a direct implementation with no multiplexing of resources. The three RAMs of the filters are accessed with one access per clock cycle. It results in a limitation of the pipeline rate of twelve cycles per data produced. Secondly, the analysis of the results can be divided in two points. The first point concerns the different approaches of the used tools. For SystemC and Handel-C tools, pipeline needs to be explicitly described. The C-code is functional with no specific programing with the ImpulseC tool. The number of stages of the pipeline is not precisely controlled with ImpulseC but indirectly through timing constraints. The automatic exploration of different options and constraints is the only solution to obtain the best compromise between the different constraints as the impact of the throughput/latency of the pipeline on the area/frequency is not straightforward. The difference of throughput between a pure sequential solution and a fully pipeline solution can be more than two orders of magnitude. This is the main source of performance/area

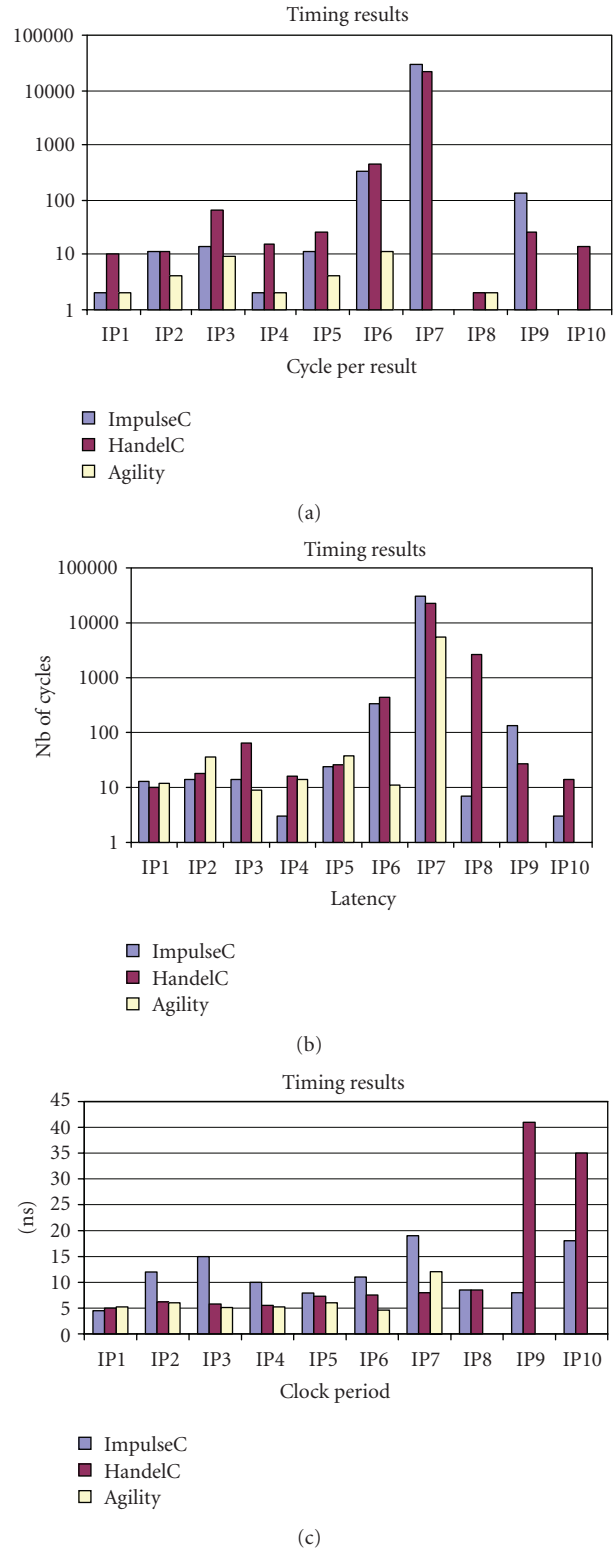


Figure 5: Timing results for throughput, latency, and clock period.

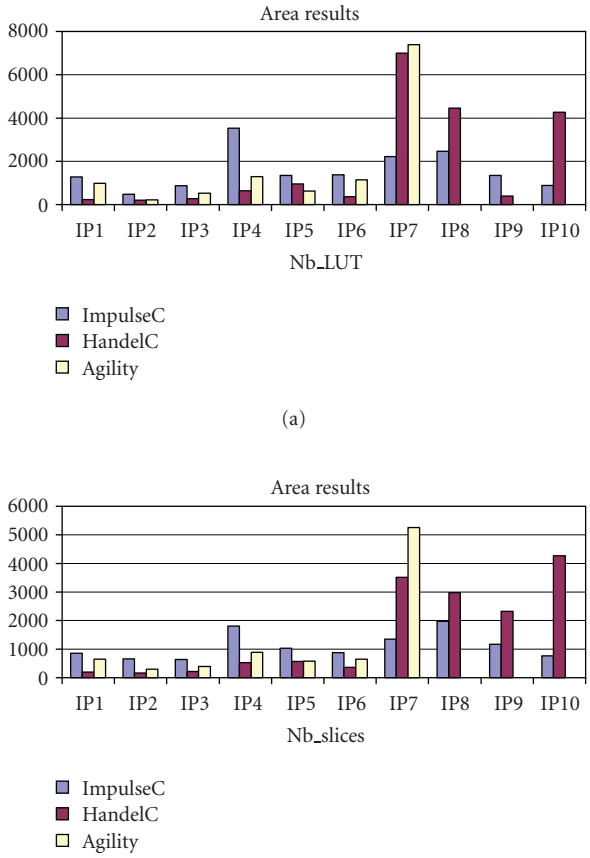


Figure 6: Area results for logic elements.

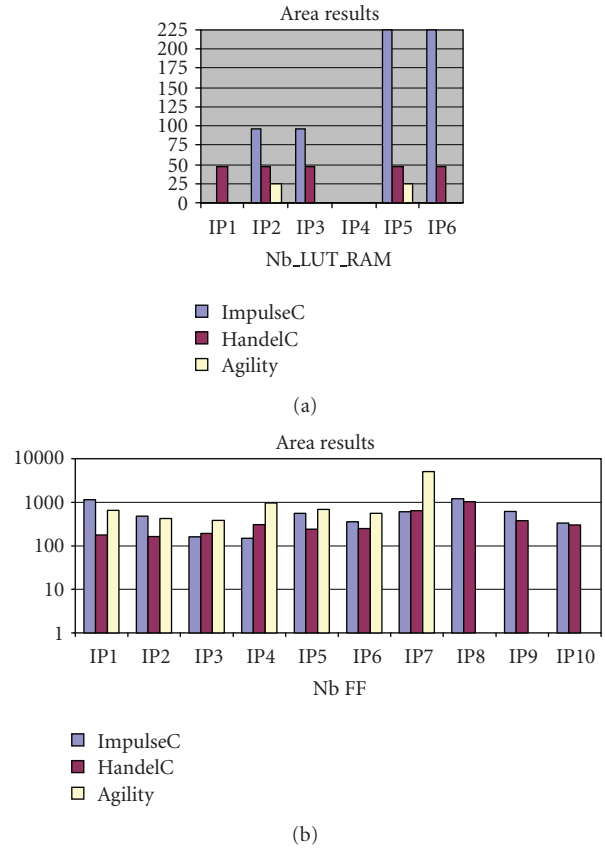


Figure 7: Area results for storage elements. (a) Number of LUT BRAM (the number is zero for IP 7 to IP 10) and (b) number of flip flop.

tradeoff at this level. This difference is increased with the implementation variability.

4.4. Area results

The area results have been obtained through VHDL generation of the various case studies followed by synthesis and place and route using Xilinx XST tool. They are presented in Figures 6 and 7. Our area metrics contain various resources present in the Xilinx Virtex-4 that are slices, flip flop, look-up table, and RAM. The synthesis and place and route options used are here the default options.

A first observation is that behavioral C-based approach with ImpulseC produces not always but often more logic and storage elements and a higher clock period than the other approaches. On the other side, ImpulseC brings the advantage of abstraction and genericity. This difference is not critical as throughput and latencies are similar.

4.5. Variability of results with compilation options

The results presented in Figures 8 to 12 show a variability of timing and area results according to the options used for synthesis and place and route. This variability depends on the front-end tool used (agility, DKDesign, and ImpulseC Codeveloper). The results presented in Figures 5 to 7 should be revisited for each option. These important clock period variations up to 150% are obtained with a variation of area cost between 100 and 200 slices, that is, 10 to 20% of area variation. The impact of tool options has a significant impact on timings compared to the impact on the size. Another major observation is the variability of results between options. This variability can be more significant than the variability between front-end tools. For instance the clock period variation is about 10 nanoseconds for the pipeline example (Figures 9 and 11). Thus ImpulseC gives better results with one option, for example, option 11 but not with another option. Agility gives better results with option 1 (Figure 9). In fact the best tradeoff is found by a careful analyze of the area/timing results. The area and timing results are not always linked as we can see with configurations 19 and 11 in Figure 8. For both configurations, the clock period is small but configuration 11 provides a higher area result compared to configuration 19. These variations also exist with placement constraints. The timing results can be either better or even worst when constraining area placement. Figure 12 compares the 24 configurations exploration with (left) and without (right) placement constraints. The improvement here is at most 0.15 nanosecond on the clock period which is not significant. Results were even worst

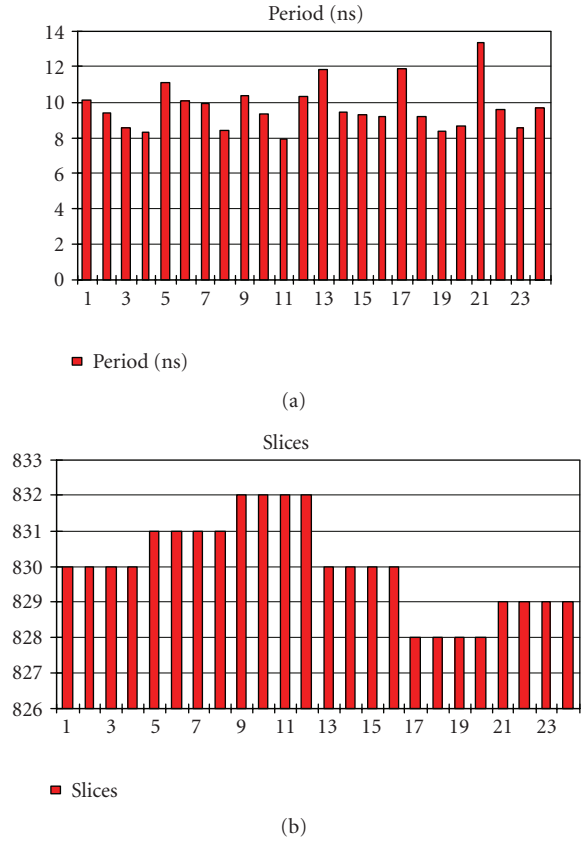


FIGURE 8: Sequential mean filter with impulseC. XST VHDL synthesis tool variation (a) period (b) slices.

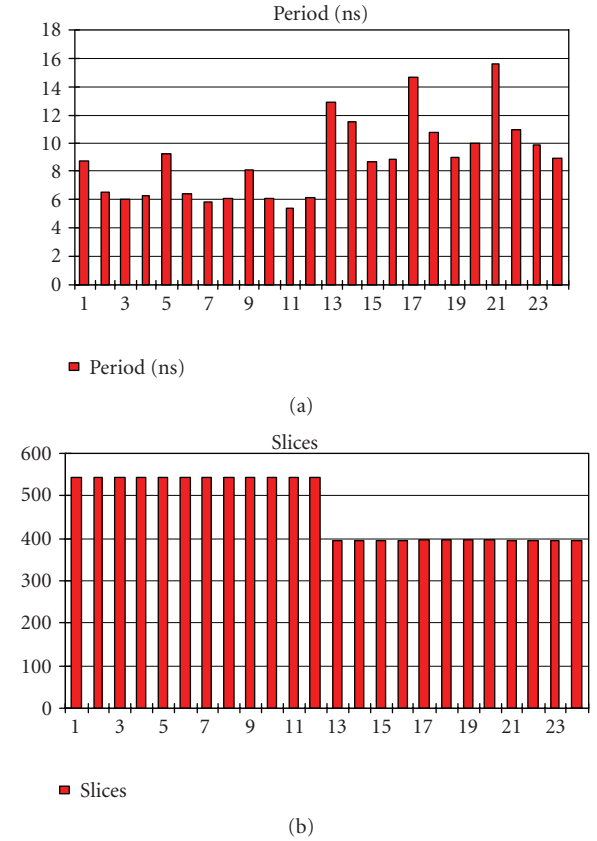


FIGURE 9: Pipeline mean filter with impulseC. XST VHDL synthesis tool variation (a) period (b) slices.

for the FFT. Thus a manual floorplanning becomes really difficult for heterogeneous hardware architectures such as FPGA. It should be reminded that obviously synthesis and place and route can incur large variations if no constraints are imposed and if large chips are selected. With large chips, the design can be spread without constraints conducting to higher delays. In our case, first the synthesis and place and route stages are done without constraints. Then a constrained place and route is used for the other configurations tested. The results are better with constraints.

The last point concerns the impact on place and route described on several examples in Figures 13 to 16 from best to worst. Best solutions are less spread and thus have reduced delays and higher operating frequencies.

The variations in terms of area and resources are obvious from the above figures. This points out that C-based synthesis may generate very different implementations resulting from C-based high-level modeling and the strong impact of backend tools. It should be noted that with heterogeneous devices such as Virtex-4 where hard cores are embedded the place and route tools may decide to implement a function in the vicinity of such embedded cores even if no interaction exists. This can be easily observed in Figures 13 and 14 (right part). This affects the quality of the implementation as the logic is spread out on the circuit. This clearly shows that there is a missing link between the system modeling level and the physical implementation. A feedback is necessary to help joint optimization.

4.6. Discussion

The C-to-hardware compilers considered here take two approaches to concurrency. The first approach chosen by Handel-C and agility compiler adds parallel constructs to the language. It forces the programmer to expose most concurrency that is not a difficult task in major cases. Handel-C provides specific constructs that dispatch collections of instructions in parallel. These additional statement constructs can be used by any programmer. The compilers considered here use a variety of techniques for inserting clock cycle boundaries. Handel-C and agility use fixed implicit rules for inserting clocks and are very simple. Assignments and delay statements each takes one cycle in Handel-C and instructions between two wait() statements take one cycle in agility SystemC. All the instructions inserted in a *par* statement are executed in one clock cycle in Handel-C. For all the implemented filters, adding manually parallelism is an easy task that can be achieved by any programmer. On the other hand, pipeline extraction can become a tricky task as algorithm must be written in that way. An example was the FFT algorithm implementation: adding pipeline from a sequential code can take a long time and changes

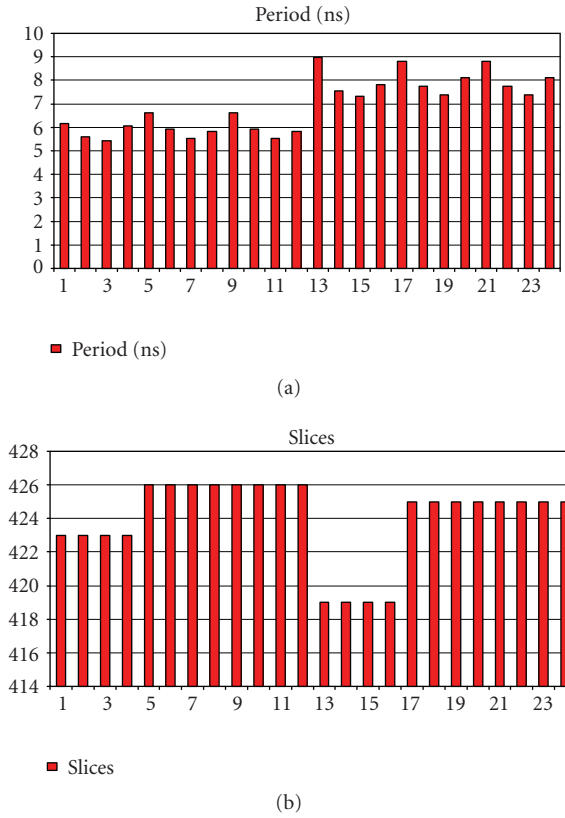


Figure 10: Sequential mean filter with agility. XST VHDL synthesis tool variation (a) period (b) slices.

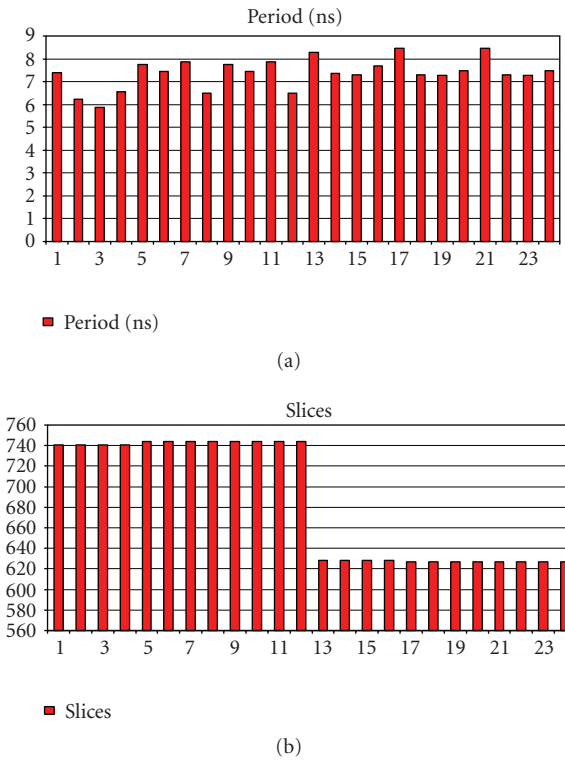


Figure 11: Pipeline mean filter with agility. XST VHDL synthesis tool variation (a) period (b) slices.

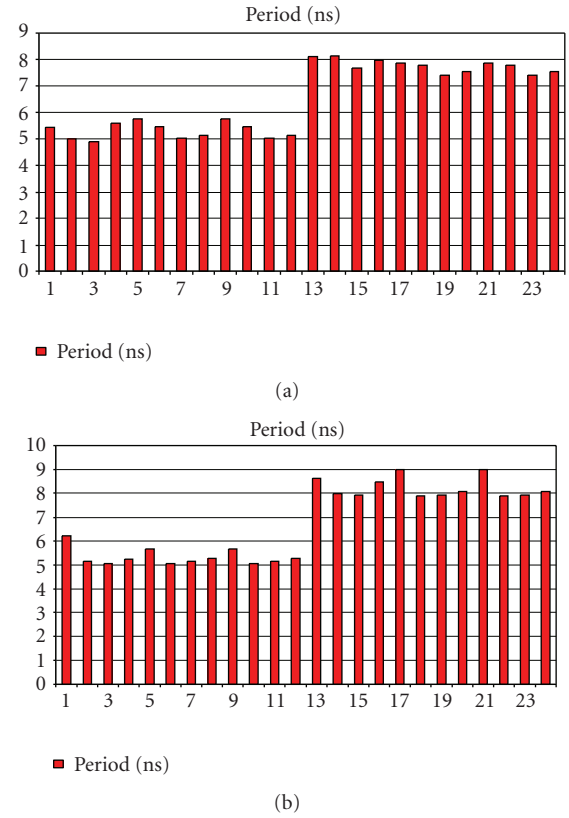
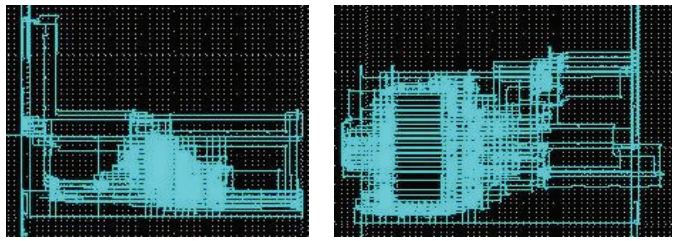


Figure 12: Pipeline median filter with agility. XST VHDL timing variation with and without placement constraints.

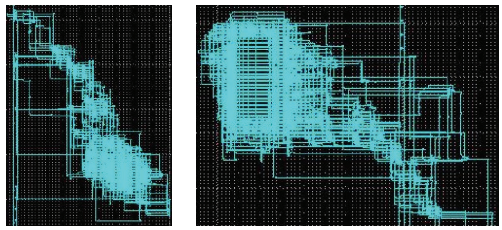
are important to make. It is even more difficult to express pipeline with Handel-C than SystemC as dependencies between instructions impose the use of different cycles. The precise control of logic/operators per clock cycle is difficult with Handel-C: either all the instructions in one stage are independent and the pipeline clock can be of one clock cycle per result or reuse is possible that makes the number of clocks per result proportional to the reuse rate. Another solution is to use a higher frequency and divide the processing in elementary cycles (one per code line). SystemC agility compiler representation becomes therefore almost an RTL level representation allowing optimization at the clock cycle level.

The second approach lets the compiler identifies parallelism helped with pragmas in the source code. ImpulseC compiler allows automatic pipelining through pragmas but only for inner loops. Loop unrolling is used to obtain full pipelining. Precise control of the number of stages is difficult with such pragmas. These simple rules can make it difficult to achieve a particular timing constraint. It is difficult to predict in advance when a second input data can be inserted, that is the throughput. Several synthesis cycles must be operated to converge to the best solution. The tool helps this exploration by automating the use of VHDL synthesis tool in the loop. Pipeline exploration is conducted automatically with VHDL synthesis on different solutions providing a frequency graph function of the latency/rate of the pipeline. This helps to obtain the higher rate/latency pipeline but with



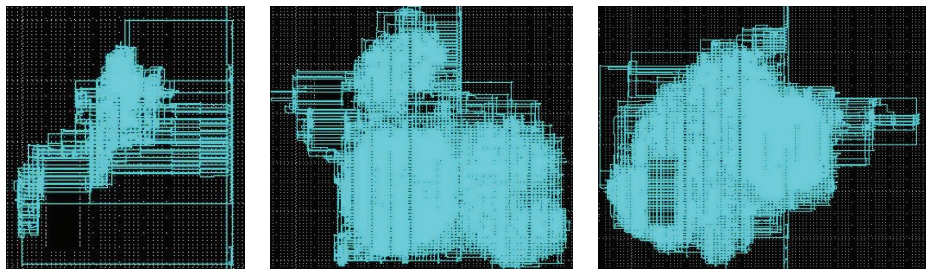
(a) (b)

Figure 13: Sequential mean filter. Place and route variations from best (a) to worst (b).



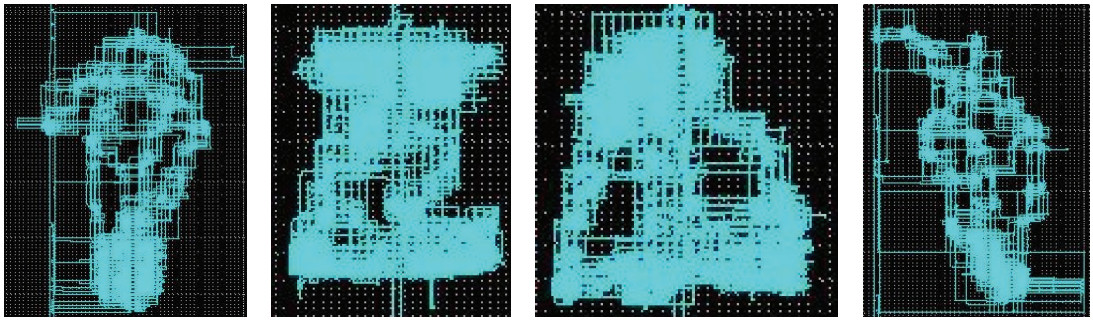
(a) (b)

Figure 14: Pipeline mean filter. Place and route variations from best (a) to worst (b).



(a) (b) (c)

Figure 15: Pipeline FFT. Place and route variations from best (left) to worst (right).



(a) (b) (c) (d)

Figure 16: Pipeline median filter. Place and route variations from best (left) to worst (right).

no considerations of the area. It is thus difficult to make a compromise between timing and area constraints.

The IP interfaces provided by ImpulseC are FIFO or memory which is better adapted to stream processing. In fact, it is difficult to design specific protocols at RTL level. Also considering local memory storage, RAM/register inference selection is only obtained through a compilation option of ImpulseC, that is for all the design and not

separately for each array, which is really limiting as registers are a limited resource in FPGA. The two other tools provide pragmas to define precisely which way to store arrays of data, with registers or RAM.

According to the data types, C-based design tools considered several approaches. The first approach neither modifies nor augments C's types but allows the compiler to adjust the width of the integer types outside the language. The second approach is to add hardware types to the C-language. Handel-C and ImpulseC compilers chose the data customization. The programmer cannot cast a variable or expression to a type with a different width, that makes the code more difficult to write. For the filter implementation, arrays are of several sizes and the indexes' sizes are different. The programmer must often use the concatenation operator to zero pad or sign extend a variable to a given width that makes the programing time longer.

Most of the Handel-C debugging time was spent in adjusting the size of data and manually programing the pipeline optimization. The programmers must carefully analyze the code to specify all the widths and it can quickly be tiresome. A parser for automatic adjusting of the size of any used variable according to the type of the operators used can be considered.

One main argument to choose an approach to use is the level of description needed at the interface level of the design. If FIFO or memory-like protocols are sufficient, behavioral C-based HLS is now a mature solution with equivalent performances and area results than a more precise almost RTL level C-based solution like agility compiler or Handel-C. Furthermore, behavioral C-based HLS provides abstraction and genericity of the pipeline allowing easy retargeting of hardware in different timing/area constraints without redesigning. This criterion is fundamental in streaming applications where throughput is the key performance parameter.

5. CONCLUSIONS AND FUTURE WORK

We have conducted a case study on the evaluation of C-based high-level synthesis systems. The objective was to assess a potential higher use of this in-area constrained high-level system multiobjective partitioning and how system decisions could be impacted. Indeed, although growing system complexity calls for high-level abstract modeling, it is still mandatory to take into account precise implementation feedback to improve performances. This puts into question the capacity of C-based tools to meet this challenge. Evidences on case studies of significant result variations among the high-level synthesis tools and their emphasis through physical synthesis options exploration challenge the use of C-based multiobjective modeling methodologies for system design. In a multiobjective approach, area-performance system design tradeoffs should be based on as accurate as possible data otherwise inappropriate design decisions could be made. We argue that implementation issues (area, frequency, and floorplan) for large scale complex systems should be taken into account when using C-based high-level modeling since currently the tools do not guarantee that high-level concurrency semantics properties are preserved. Indeed, extracted concurrency at high level is challenged through code and representation transformation as well as resources constraints.

Solution to this comes through an integrated flow with concurrency properties propagated as constraints as well as concurrency feedback to the highest level.

Future work will extend the size of the case studies and automate the evaluation process.

ACKNOWLEDGMENT

The authors are grateful to the reviewers for their excellent reviews which helped improve the paper dramatically.

REFERENCES

[1] ITRS 2007 Edition—System, http://www.itrs.net/.

[2] IEEE 1666 Standard SystemC Language Reference Manual, http://www.systemc.org/.

[3] C. Haubelt, J. Falk, J. Keinert, et al., "A systemC-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 47580, 22 pages, 2007.

[4] S. Ouadjaout and D. Houzet, "Generation of embedded hardware/software from systemC," *EURASIP Journal on Embedded Systems*, vol. 2006, Article ID 18526, 11 pages, 2006.

[5] M. O. Cheema, L. Lacassagne, and O. Hammami, "System-platforms-based systemC TLM design of image processing chains for embedded applications," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 71043, 14 pages, 2007.

[6] D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs," in *Proceedings of the 3rd IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '95)*, pp. 136–144, Napa Valley, Calif, USA, April 1995.

[7] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, "Cyber"," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '99)*, pp. 390–393, Munich, Germany, March 1999.

[8] D. C. Ku and G. De Micheli, "Hardware C: a language for hardware design," Tech. Rep. CSTL-TR 90-419, Computer System Laboratory, Stanford University, Stanford, Calif, USA, August 2000.

[9] T. Kambe, A. Yamada, K. Nishida, et al., "A C-based synthesis system, Bach and its application," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '01)*, pp. 151–155, Yokohama, Japan, January-February 2001.

[10] E. Grimpe and F. 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 (CODES+ISSS '03)*, pp. 25–30, Newport Beach, Calif, USA, October 2003.

[11] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *Proceedings of the 16th International Conference on VLSI Design*, pp. 461–466, New Delhi, India, January 2003.

[12] V. S. Saun and P. R. Panda, "Extracting exact finite state machines from behavioral systemC descriptions," in *Proceedings of the 18th International Conference on VLSI Design*, pp. 280–285, Kolkata, India, January 2005.

[13] H. D. Patel, S. K. Shukla, and R. A. Bergamaschi, "Heterogeneous behavioral hierarchy extensions for systemC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 4, pp. 765–780, 2007.
[14] S. Chtourou and O. Hammami, "SystemC space exploration of behavioral synthesis options on area, performance and power consumption," in *Proceedings of the 17th International Conference on Microelectronics (ICM '05)*, pp. 67–71, Islamabad, Pakistan, December 2005.
[15] ROCCC, http://www.cs.ucr.edu/~roccc/.
[16] Synopsys, "Behavioral Compiler User Guide Version 1999.10," 1999.
[17] Agility,http://www.agilityds.com/products/default.aspx.
[18] Forte Design, http://www.forteds.com/.
[19] Orinoco Dale, http://www.chipvision.com/company/index.php.
[20] ImpulseC Inc, "Co-developper's user guide," 2007, http://www.impulsec.com/.
[21] CatapultC, http://www.mentor.com/.
[22] S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Design and Test of Computers*, vol. 23, no. 5, pp. 375–386, 2006.
[23] Handel-C, http://www.celoxica.com/.
[24] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, 2007.
[25] J. Revelles, C. Ureña, and M. Lastra, "An efficient parametric algorithm for octree traversal," in *Proceedings of the 8th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG '00)*, Plzen-Bory, Czech Republic, February 2000.
[26] Xilinx Virtex-4, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.
[27] Xilinx ISE 9.2, http://www.xilinx.com/ise/logic_design_prod/foundation.htm.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 519474, 14 pages
doi:10.1155/2008/519474

Research Article

Model-Driven Validation of SystemC Designs

Hiren D. Patel[1] and Sandeep K. Shukla[2]

[1] *Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 545K Cory Hall, Berkeley, CA 94720, USA*
[2] *Formal Engineering Research with Models, Abstractions and Transformations, Virginia Tech., 302 Whittemore Hall, Blacksburg, VA 24060, USA*

Correspondence should be addressed to Hiren D. Patel, hiren@eecs.berkeley.edu

Received 1 October 2007; Revised 24 February 2008; Accepted 7 April 2008

Recommended by Axel Jantsch

Functional test generation for dynamic validation of current system level designs is a challenging task. Manual test writing or automated random test generation techniques are often used for such validation practices. However, directing tests to particular reachable states of a SystemC model is often difficult, especially when these models are large and complex. In this work, we present a model-driven methodology for generating directed tests that take the SystemC model under validation to specific reachable states. This allows the validation to uncover very specific scenarios which lead to different corner cases. Our formal modeling is done entirely within the Microsoft SpecExplorer tool to formally describe the specification of the system under validation in the formal notation of AsmL. We also exploit SpecExplorer's abilities for state space exploration for our test generation, and its APIs for connecting the model to real programs to drive the validation of SystemC models with the generated test cases.

1. INTRODUCTION

SystemC [1] has gained considerable traction in the electronic design automation community as an entry language for system-level modeling, simulation, and validation. It is often used in the early stages of a product's design cycle for fast design space exploration. This means that an ample amount of time is spent in simulating and validating the system-level design. The standard procedure for validation requires a designer to generate a set of input sequences that exercise all the relevant and important properties of the system. Randomized test generation and simulation are almost always augmented with manual test sequences for properties that are critical to the functionality of the system. Manually coming up with the input sequences that validate properties is an arduous and a difficult task that needs the designer to know the complete details of the design. Let us take a simple example such as a FIFO component ignoring inputs when it is full. The input sequences for testing this property must ensure that the design transitions into the state whenthe FIFO is full. However, there are multiple paths that can take the design into the state where the FIFO is full. So, it is important to generate input sequences that cover the trivial and nontrivial cases. Moreover, it is crucial that the designer can direct the validation procedure for the design. In our work, we focus on validating the design by directing the generation of tests. It is not always simple to come up with these input sequences. This is particularly true for large and complex designs where it is difficult to identify and describe the input sequences to reach a particular state [2, 3]. Furthermore, there may be multiple paths (different input sequences) that transition the system to the same state. Authors in [2] provide input sequences by performing static analysis on the SystemC source and then use supporting tools for generating the sequences that serve as tests for the design. This approach is limited by the strength of the static analysis tools. In addition, it does not provide the flexibility for describing the reachable state of interest. Also, static analysis requires sophisticated syntactic analysis and the construction of a semantic model, which for a language like SystemC built on C++ is difficult due to the lack of a formal semantics for it. In fact, [2, 3] do not precisely describe a semantics for SystemC. It is also difficult to diagnose the exact transition that causes a failed test-case execution. For this reason, it is

important to provide designers with a methodology and set of tools that ease the burden of validation and in particular directed test-casegeneration and diagnosis.

SpecExplorer is a tool developed by Microsoft for model-based specification with support for conformance testing. The essence of SpecExplorer is in describing the system under investigation as a model program either in AsmL [4] or Spec# [5] and performing conformance tests on an implementation model in the form of some software implementation. The model program in AsmL serves as a formal specification of the intended system because AsmL employs the abstract state machine (ASM) [6, 7] formalism. From here on, we use "specification" interchangeably with "model program." ASMs allow for specifying hardware or software systems at the desired level of abstraction by which the specification can focus on only the crucial aspects of the intended system. The added quality of ASM specifications being executable aids in verifying whether the specification satisfies the requirements, whether the implementation satisfies the specification, and transitively whether the implementation satisfies the requirements. This makes ASMs and SpecExplorer a suitable formalism and tool, respectively, for semantic modeling, simulation and as we show in this paper, for exploration, and test-case generation.

Previous works in [8–10] use SpecExplorer to put forward a development and verification methodology for SystemC. Except their focus is on the assertion-based verification of SystemC designs using Property Specification Language (PSL). Their work mentions test-case generation as a possibility but this important aspect of validation was largely ignored.

In this work, we present a model-driven methodology not only specifying and developing, but also validating system-level designs for SystemC [11]. Figure 1 shows a block level schematic of the model-driven methodology. We create an abstract model from a natural language specification and this abstract model is what we call the semantic model. It is specified using AsmL (an implementation language for ASMs). Since SystemC follows the discrete-event (DE) simulation semantics, we provide an AsmL specification for the DE semantics such that designers can mimic the modeling style of SystemC while using AsmL for their intended design. The specification for the intended system can then be executed with the DE specification. For testing whether an implementation model satisfies the specification, SpecExplorer provides tools for exploration and test-case generation. Exploration results in an automaton from the specification. This automaton is then used to generate tests. However, SpecExplorer only allows bindings to C# and Visual Basic implementation models, but for our purpose we require bindings to the implementation model in SystemC. To do this, we provide two wrappers that export functions of the SystemC library and the implementation model to libraries that are used in SpecExplorer. These functions are used in a C# interface wrapper and then bound in SpecExplorer. We use SpecExplorer's exploration and test-case generation to create tests that are directed to reach interesting states of the design. While the input sequences are executed on the semantic model, the same inputs are exercised on the SystemC implementation model to validate whether the SystemC implementation model conforms to the AsmL specification.

1.1. Main contributions

This work presents a methodology for specification, model-driven development, and validation of SystemC models. This methodology is based on Microsoft SpecExplorer. In this paper:

(i) we provide a formal semantics for the most recent version of SystemC's DE simulation using ASMs as the semantic foundation,

(ii) we leverage Microsoft's implementation of ASMs operational semantics to provide a simulation and debugging framework for the design specification described in AsmL,

(iii) we again leverage Microsoft's existing algorithms for state space exploration in SpecExplorer to be effectively used for SystemC test generation for finding corner cases in the SystemC implementation,

(iv) we show how to generate wrappers for the SystemC library and SystemC implementation model to allow SpecExplorer to drive the SystemC simulation through C#,

(v) we show how to do directed test generation and diagnosis for bug finding such that oversights and true errors in the implementation are found using SpecExplorer's test generation tool.

2. OUTLINE

The outline of this article is as follows: Section 3 discusses the necessary background and related work in fully understanding the proposed methodology and Section 4 presents the design flow. We present examples in Section 5 followed by our experience in using this approach in Section 6, and finally conclude in Section 7.

3. BACKGROUND AND RELATED WORK

3.1. Abstract state machines

Abstract State Machines (ASMs) [6, 7] are finite sets of transition rules. A transition rule consists of a guard and an action. A transition rule looks like *if Guard then Updates* where the "Guard" evaluates to a Boolean value and "Updates"is a finite set of assignments. The set of assignments update values of variables of the state. These assignments are depicted as

$$f(t_1, \ldots, t_n) := t_{0,} \tag{1}$$

where t_0 to t_n are parameters and f denotes a function or a variable (a 0-ary function). At each given state (also referred to as a step), the parameters t_0 to t_n are evaluated first to obtain their values denoted by v_k for $k = \{0, \ldots, n\}$,

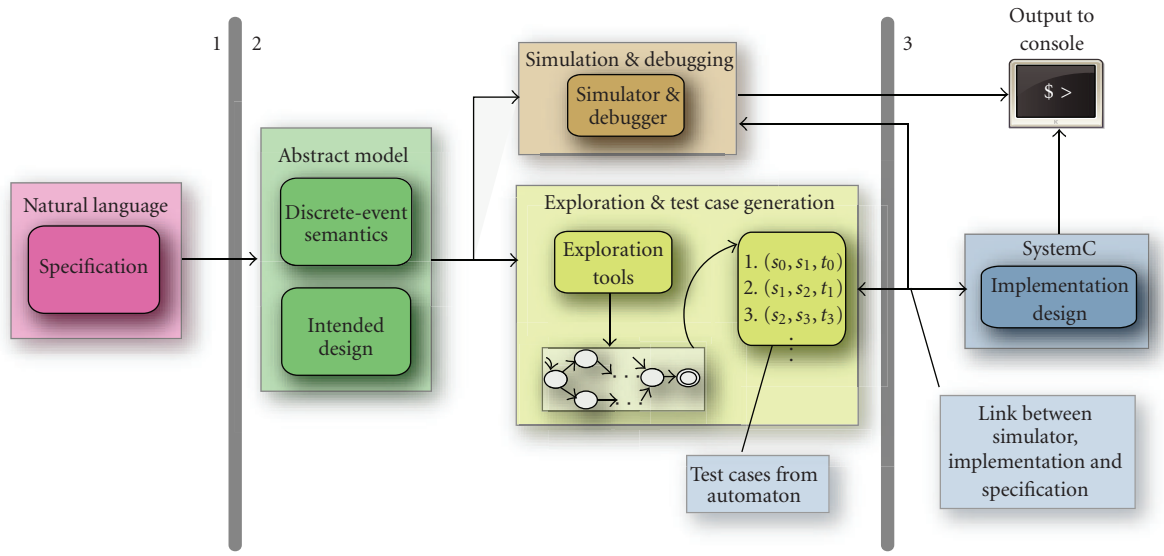


Figure 1: Methodology overview.

respectively. Upon the evaluation of v_k, the functions in the Updates set are evaluated. Hence, $f(v_1, \ldots, v_n)$ is updated to v_0. A run of an ASM simultaneously fires all transition rules whose guard evaluates to true in that state.

There are several ASM variants whose basis is the standard ASM as described above. Examples of the variants are Turbo-ASMs [6], synchronous ASMs, and distributed/asynchronous ASMs [6, 12]. Each of these variants posses certain descriptive capabilities. For example, Turbo-ASMs are ASMs appropriate for parallel and sequential composition, and recursive submachine calls.

3.2. SpecExplorer

SpecExplorer is a specification exploration environment developed by Microsoft [4, 13] that is used for model-based specification and conformance testing of software systems. The language used for specification can be either AsmL [14, 15], a derivative of ASMs, or Spec# [5, 16].

Exploration in SpecExplorer generates automata from the specification. This requires indicating the actions and states of interest. The four types of actions supported are controllable, observable, scenario, and probe. Controllable actions are those that the specification invokes and the observable are the ones invoked by the environment. A scenario action brings the state elements to a particular starting state and the probe action is invoked at every state in efforts to look into the implementation. There are a number of exploration techniques that can be used to prune the state space. For example, state filters are Boolean expressions that must be satisfied by all explored states. Similarly, the state space can be limited to a set of representative state elements as well as state groupings. Finally, there is support for generating test cases from the automaton generated from the exploration. SpecExplorer supports coverage, shortest path, and random walk test suite generation. These are discussed in further detail in the reference documents of SpecExplorer [4].

3.3. Discrete-event semantics using ASMs

Our discrete-event semantics specification uses turbo ASMs with object orientation and polymorphism capabilities of AsmL. We specify the simulation semantics in the `discrete-event` class shown in Figure 2. The numbers associated with each function shows the order in which these are executed. For example, after the instantiations of the variables in (1), the entry point is the `trigger discrete-event` function marked with (2). Note that we use AsmL's parallel constructs such as the `forall` in `trigger behaviors` to specify parallel execution of the behaviors. This corresponds well *to the unspecified execution order of the processes* in SystemC's simulation semantics. We have created additional classes that help the designer in describing a semantic model in AsmL so that it follows our discrete-event simulation semantics. Examples are the `denode` class that represents the behavior of a specific process and the `degraph` that represents the netlist of the entire design. Anyhow, we do not present these here because the focus here is test generation. We have made our entire semantics and examples downloadable via the web at [17].

The simulation begins with a function `start` that is not shown in Figure 2. This function updates the state variables `stopTime` and designgraph that hold the duration of the simulation and a structural graph of the system being modeled, respectively. After the values of these two variables are updated, the simulation begins by invoking `trigger discrete-event`. This initializes the simulation by triggering every behavior in the system that in turn generates events. After this, the simulation iterates through the `evaluate`, `update`, and `proceedTime` functions. In `AsmL until fixpoint` only terminates when there are no updates available or there is an inconsistency in which

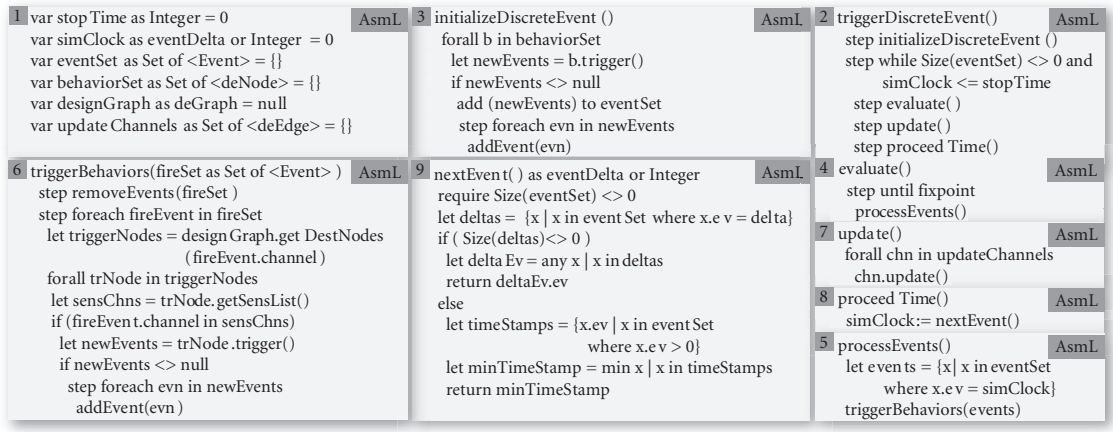


Figure 2: Discrete-event semantics using AsmL.

the simulation throws an exception. The `processEvents` function retrieves a set of events that match `simClock` and these events are triggered via `triggerBehaviors`. The `update` function invokes an update on all specified channels very much like SystemC and the `proceedTime` forwards the simulation time by calling `nextEvent`. The `nextEvent` returns an event from the event set. The semantics of this enforce the processing of all delta events first before a timed event.

3.4. Exploration in SpecExplorer

SpecExplorer provides methods for generating automata based on the designer's input for exploring the specification. The designer specifies the actions of interest in the exploration settings, for an FSM to be generated. These actions are functions in the specification. They can be classified into four types: controllable, observable, scenario, and probe [4]. Controllable typed actions are the functions that are invoked by SpecExplorer and observable are the actions that SpecExplorer waits for a response from the implementation model. Probe actions simply query for state information, and scenario actions provide a way of reaching a starting state for the exploration. Selectively exploring transitions of the specification is possible via a variety of methods such as parameter selection, method restriction, state filtering, and state groupings. We direct the reader to [4, 18] for further information regarding exploration techniques in SpecExplorer.

Accepting states describe the states at which a test must finish. These accepting states are defined using a state-based expression. For example, a simple `true`suggests that all states are accepting states and `FULL = true` suggests that only the states where the state variable `FULL` is true are accepting states. The test case generator uses this state-based expression and computes all possible paths from the initial state to the accepting states given that all other constraints such as state filters are satisfied. Our methodology uses the accepting states and methods for selectively exploring transitions for directing the test-case generation and diagnosis.

A work on functional verification of SystemC models is proposed in [2]. In general, an FSM is generated by performing static analysis on the source code very much like [9] and this FSM is used to generate test sequences for the system under investigation. Authors of [2] use an error simulation to inject errors into a model that is compared with an error-free model for detecting possible errors. The biggest difference in the approach described in [2] is the lack of control a designer has in directing the test-case generation. For instance, the final states (accepting states for us) are not defined by the designer. Full state space exploration is often times not practical and it is essential to be able to better direct the exploration. In addition, these authors use static analysis to parse SystemC and generate extended FSMs, but they do not provide formal semantics for SystemC. This is important to check whether the abstract model in extended FSMs is a correct representation of the SystemC design. Our work on the other hand focuses on providing designers with the capability of defining the exact states of interest and only generating input sequences up to those states. This is done by creating two independent models, first the semantic then the implementation. Then the semantic model is used for directing tests in the implementation model.

Recent work in design and verification methodologies for SystemC using SpecExplorer are presented in [8–10]. In their approach, the design begins with UML specifications for the SystemC design and PSL properties. Both the SystemC and PSL UML specification are translated into ASM descriptions and simulated using SpecExplorer's model simulator. Reference [9] also implements an AsmL specification of SystemC's simulation semantics [8], however, we were unable to obtain an executable version for us to reuse. This simulation provides a first stage of verification that ensures that the PSL properties specified hold in the ASM specification of the SystemC design. This SystemC ASM specification is then translated into a SystemC design using C++ and the PSL properties in C#. These two are then compiled and

later executed together to verify whether the properties are satisfied at the low-level implementation.

In summary, their work uses the modeling, simulation, and automata generation capabilities of SpecExplorer but not the test-case generation and execution tools. They only hint towards the possibility of using it for test generation in their overall design flow. In [9], the same authors present algorithms for generating FSMs from SystemC source code. They have translation tools to convert the extracted FSM into AsmL specification. These algorithms for FSM generation use similar concepts such as state grouping as the ones in SpecExplorer, and once again the authors hint that this can be used for conformance testing and equivalence checking, but this is not presented. Their work attempts at providing a top-down and bottom-up flow for verification of SystemC designs mainly focusing on PSL assertion verification, but not test-case generation and execution for validation purposes. The main distinction of their work and our work is that we do not convert ASMs to SystemC or SystemC to ASMs. Instead, we promote a model-driven approach where the semantic model is done first for the correctness of the functionality and conformance to the natural specification. This is followed by an implementation model in SystemC developed independently. Then, the conformance between the semantic and the implementation model is validated by generating tests in SpecExplorer and executing them in both the semantic and the implementation model. This is how the work in this paper distinguishes itself from the works mentioned in [8–10].

Authors of [3] propose labeled Kripke structure-based semantics for SystemC and predicate abstraction techniques from these structures for verification. They treat every thread as a labeled Kripke structure and then define a parallel composition operator to compose the threads. They also provide formal semantics to SystemC. Our work differs from this work in the same way that of [2] that we provide a model-driven approach. The authors of [3] create their abstract model from the implementation. Moreover, they do not present any algorithms for traversing the parallely composed Kripke structures for test generation.

The authors in [12] presented the ASM-based SystemC semantics that was later augmented for the newer versions of SystemC by [10]. However, the original ASM semantics in [12] did not present any support for test-case generation and validation from ASMs and was designed for an older version of SystemC.

4. DESIGN FLOW

The necessary components in employing this design flow are shown in Figure 3. We separate these components into four phases. These phase separations also describe the steps that a designer takes in using this methodology. Note that modular development of the system is possible and recommended, but in our description here, we assume the designer fully describes the system in each phase before proceeding to the next. This is done here to describe the methodology concisely and transparently. We also annotate Figure 3 with functions specific for a hardware FIFO component as our intended system (and from here on referred to it as that). We describe this example in detail in Section 5.

4.1. Semantic modeling and simulation

A typical usage mode of this methodology starts in phase `A`. In this phase, the designer creates the semantic model using constructs introduced by our discrete-event simulator in AsmL. For example, our semantic model of the FIFO component contains class declarations for the FIFO component, a test driver and a clock. The modeling is fashioned to be similar to SystemC's modeling style so that there can be a direct mapping from the specification to the implementation. We show two of the important functions invoked by the driver that provide stimulus for the system. They are `WriteRequest` and `ReadRequest`. These functions also specify the contract between the specification and the implementation model during test-case generation. Once the semantic model is specified in AsmL, SpecExplorer's model simulator is used to validate the semantic model by simulation. One can also use some of the model checking links of SpecExplorer but we did not explore that possibility for now.

4.2. SystemC modeling, simulation, and wrappers

4.2.1. Modeling

The second phase `B` is where the designer implements the FIFO component in SystemC, which we call the `implementation model`. It also contains some of the same functions that are listed in the `semantic model` in phase `A`. This is a result of having a clear specification of the system before moving to an implementation model. Thus, the designer has already captured the responsibilities of the members in the specification. These specification contracts translate to the implementation model.

4.2.2. Simulation

The implementation model is simulated with the OSCI SystemC [1] simulator and if required, any other supporting tools may be used. This is standard practice when developing systems in SystemC.

4.2.3. Wrapper generation

SystemC wrapper

From the SystemC library, the only function that has to be exported is `sc_start()`. This function is responsible for executing the SystemC simulation. Note that this is done only once and then can be reused, because this same exported function is used for executing the simulation for different implementation models. However, we require the SystemC library to be a static library. This enables invocations of exported functions from a C# program. This does not incur any changes to the original source code but instead, we declare the `sc_main()` in an additional source code file to allow for static compilation.

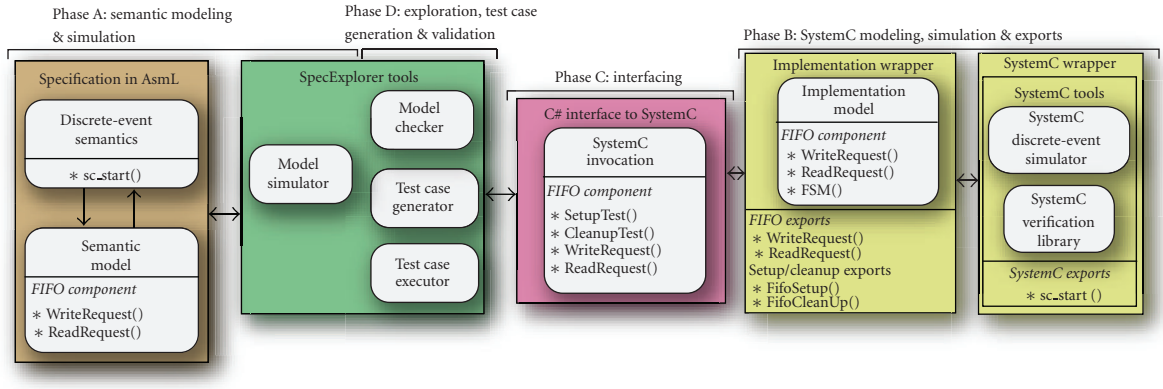


Figure 3: Modeling, exploration, and validation design flow.

Implementation model wrapper

For the implementation model, the designer must be aware of the functions that stimulate the system. This is because the test cases generated in SpecExplorer will invoke these functions to make transitions in the implementation model. For the FIFO component, the designer may have implemented a driver component during the modeling and simulation to control the signals for write and read requests. These inputs stimulate the FIFO component and should be the signals that are toggled during the execution of the test cases. Therefore, we first add global functions `WriteRequest` and `ReadRequest` that change the value on the respective write and read signals and then export these functions. In addition, we export two functions necessary for setting up and cleaning up the simulation denoted by `FifoSetup` and `FifoCleanUp`. The setup function creates a global instance of the FIFO component and its respective input and output signals and the clean up releases any dynamically allocated memory during the setup. Once these functions are exported, a C# program can simulate the FIFO component using the exported functions.

4.3. C# interface for SpecExplorer and SystemC

Phase `C` glues SpecExplorer with SystemC via C# such that the execution of the test cases generated by SpecExplorer symmetrically performs transitions in the implementation model. This conveniently allows the test cases to traverse the implementation model's state space as it does in the generated automaton in SpecExplorer.

The C# interface imports functions from the SystemC and the implementation model wrappers. These imported functions act as regular global functions in the C# program. We create an abstract C# class with members for the setup, cleanup, requesting a write, and requesting a read. Each of these members invoke the imported functions. This allows SpecExplorer to invoke the members in the abstract C# class that in turn invoke the functions exported in the wrappers. It is necessary for the C# class members to have the exact type signatures as described in the specification. For example, `WriteRequest` takes an input argument of integer type and returns a void, so the C# class member must have this exact same type signature. If this does not conform, then the bindings from SpecExplorer are not possible. Furthermore, the C# program must be compiled as a class library to load it as a reference in SpecExplorer.

4.4. Validation, test-case generation, and execution

The final phase `D` is where the test-case generation and execution are done. This validation phase again requires some setup but it is necessary to have the components described in phases `A` to `C` completed before proceeding with this phase. In this phase, the designer decides what properties of the system are to be tested. Based on that decision, the designer selects actions in the exploration settings and the accepting states where the tests should terminate. Then an automaton is generated. Before executing these test cases, bindings to the actions selected for exploration must be made. These actions are bound to members in the C# class library. The execution of the test cases makes transitions in the automaton generated in SpecExplorer and simultaneously stimulates the inputs in the implementation model making the respective state transitions. Inconsistencies and errors can be detected in this validation phase. SpecExplorer also shows a trace of the path taken on each test and the point at which a failed test case throws an exception. This helps the diagnosis of problems in the implementation.

5. RESULTS: VALIDATION OF FIFO, FIR, AND GCD

We present three examples of validating designs with varying complexity. The first is the FIFO component that has been used as a running example throughout the paper. This example is discussed in detail whereas the other two examples of the greatest common divisor (GCD) and the finite impulse response (FIR) are only briefly presented.

5.1. FIFO

We elaborate on the FIFO component example in this section by presenting a block diagram in Figure 4. This is a simple FIFO component parameterized by `n` that represents the size

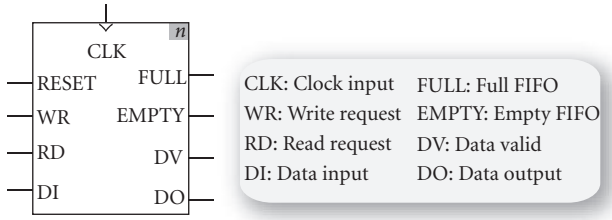


Figure 4: A FIFO component.

of the FIFO. Throughout our discussion we assume `n = 3`. The inputs to the FIFO are `WR`, `RD`, `DI`, and `CLK` and the outputs are `FULL`, `EMPTY`, `DO`, and `DV`. The `WR` requests a write onto the FIFO. If the FIFO is not full, then the data available at `DI` is stored in the FIFO. Otherwise, the data is not inserted. The `RD` input requests a read on the FIFO. This extracts the top data element and outputs it onto `DO` while raising the signal on `DV` signifying that a data is valid. `FULL` is raised high when the FIFO reaches its maximum size `n` and `EMPTY` when the number of elements in the FIFO are zero. This FIFO component is triggered on the rising edge of the clock `CLK` and every request takes two clock cycles. The `RESET` simply clears out the FIFO and again takes two clock cycles.

5.1.1. AsmL specification and systemC implementation

The AsmL specification for the FIFO component employs the DE semantics that is also described in AsmL. This is shown in Figure 5. The two basic components are the `requestGenerator` and the `FIFOBlock`. As the names suggest, the former serves as a driver for the latter. The member name of the respective classes are overlayed with a box in Figure 5. In the class declaration of `requestGenerator`, we start with the constructor that simply invokes a base class constructor of the inherited class `deNode`, which is a class available from the DE specification. The `trigger` member is in terms of SystemC terminology, the entry function of the component, or in other words, the member that is invoked by the simulator. The remaining two members, `ReadRequest` and `WriteRequest`, perform requests on the FIFO component. For the `FIFOBlock`, we again present the constructor followed by three members that perform the `Full`, `Empty`, and `Reset` operations. The entry function of the `FIFOBlock` component only invokes the internal member `FSM`.

This `FSM` member presents the crux of the FIFO component's behavior as an FSM. The FSM has three states `INIT`, `REQUESTS`, and `WAIT`. The initial state of the FSM is `INIT` after which it enters the `REQUESTS` state. Upon receiving a request of either read or write, the `FULL` and `EMPTY` states are updated via the `FULL` and `EMPTY` functions. If a write is requested, then the respective states and flags are updated and the same is done if a read was requested. The write request is given priority whenever both requests are simultaneously made. After the requests are serviced, the state of the FSM is changed to `WAIT`, which is when some of the state variable values are reset and the state is returned back to accepting requests at the next step. Note that the `WAIT` state in the SystemC implementation model could use SystemC's `wait` if a SystemC SC_THREAD process type is used. In our implementation model, we use SC_METHOD SystemC processes, and hence the need for an internal `WAIT` state. Also note that we do not present the AsmL code that instantiates and connects the two components since that is relatively straightforward. The lines marked with a ⋆ (star) are either altered or added only during exploration to prune the state space and only result in the states and transition that are desired for a particular validation scenario. We explain this in more detail in the exploration section.

5.1.2. Exploration and test cases for specific properties

Notice in Figure 6 the SystemC code that describes the FIFO's FSM implementation. We have intentionally commented out code fragments in this figure such that it is possible to cause an overflow and an underflow situation. This depicts possible modeling errors and bugs in the implementation. Now, we present test-case generation for validating whether these two basic but essential properties of the FIFO hold. The overflow situation occurs when a write is requested and when the FIFO is full. An underflow occurs when a read is requested and the FIFO is empty.

In order to generate test cases for either of the properties, we need to explore the specification. Exploration using SpecExplorer requires the use of several abstraction techniques since the semantic model may suffer from state space explosion. During this procedure, it is important to understand how the automaton is generated from ASM specifications. We forward the readers to [19] for details on the algorithms for generating automata from ASMs, but we point out that the automaton generated is based on the specified actions and states of interest. It is not generated via the execution of the entire semantic model, but simply the actions specified. The automata for the FIFO are shown in Figure 7.

We present details on some of the necessary techniques used in exploring the FIFO example. The techniques presented here are used for other properties and semantic models. However, we only present detailed discussion for the overflow property. We start by assuming that the ⋆ statements in Figure 5 are not present in the semantic model. So, for generating test cases for the overflow property, we perform `n + 1` number of successive writes without any read requests.

For successful write requests, we make `WriteRequest` and `FSM` controllable actions in the exploration configurations. This is because `WriteRequest` is responsible for issuing the request and `FSM` is the member that actually updates the internal FIFO. Since we want to essentially test for write requests until the `FULL` signal is `true`, our accepting states are those in which `FULL.read() = true`. Therefore, in the exploration settings we set the accepting states condition with the expression for showing the FIFO is full. Notice that the `WriteRequest` takes in an argument of type integer and by default SpecExplorer assigns a set of possible integers for

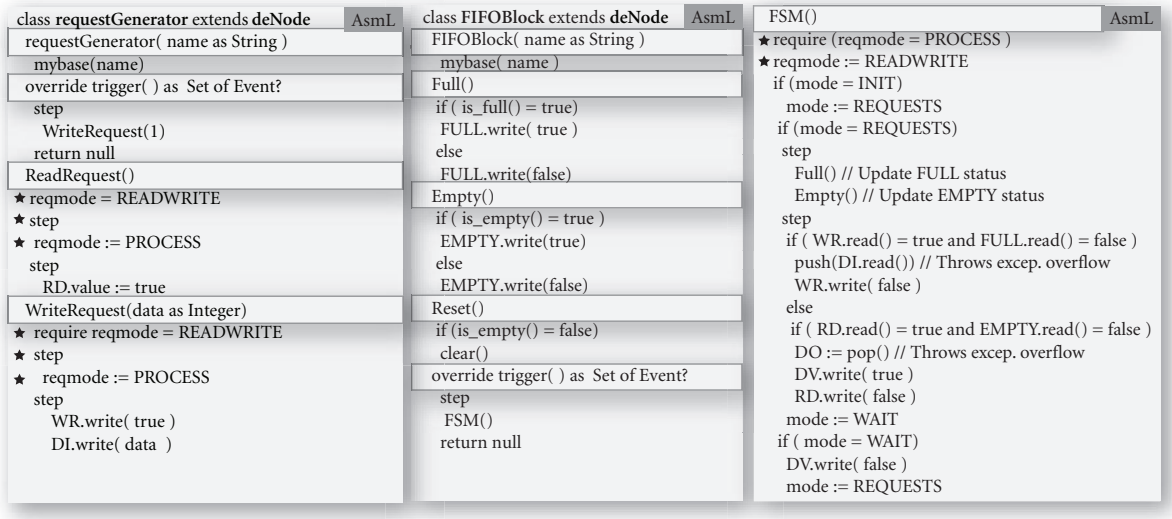


Figure 5: FIFO specification in AsmL.

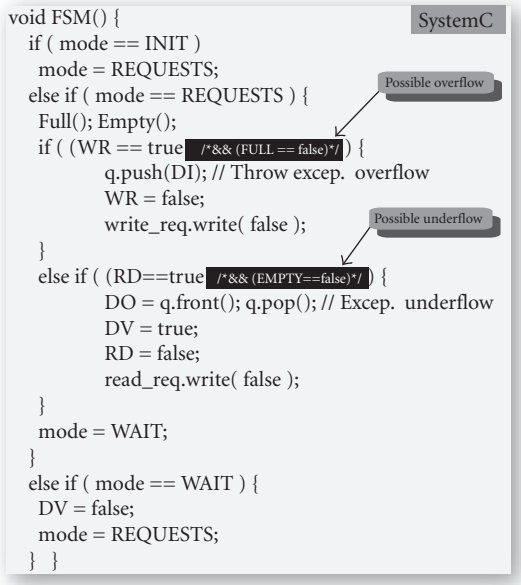
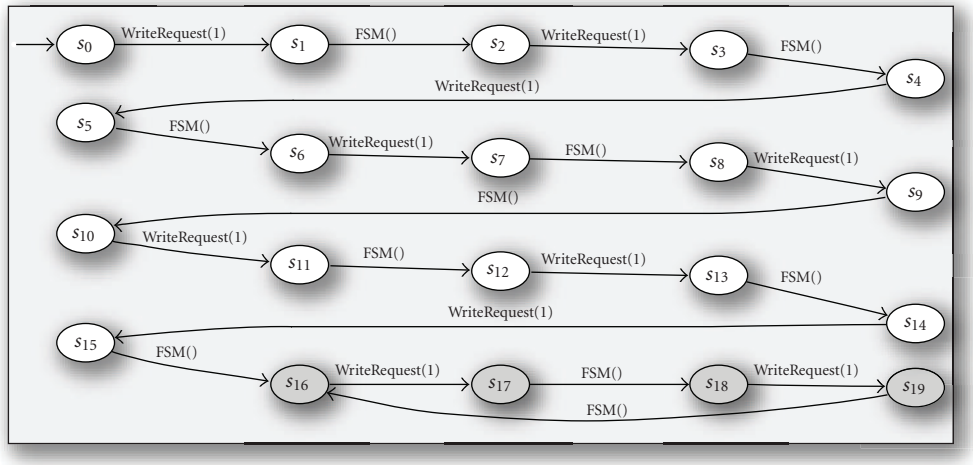


Figure 6: SystemC FIFO FSM code snippet.

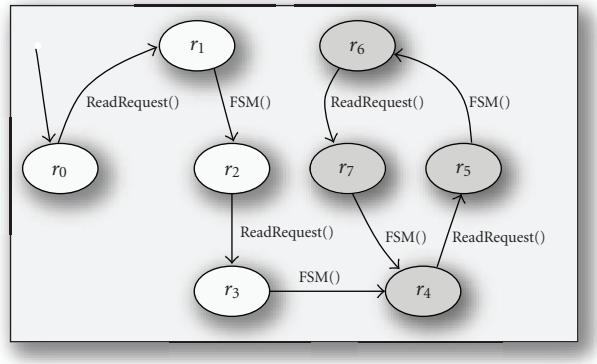
the integer type. This can be changed to whatever is desired. For simplicity, we change this default value of type integer to just `1` to avoid creating an automaton with all possible paths with all integer inputs.

This above exploration configuration results in an invocation of each action from every state. This may be the correct automaton desired by the validation engineer, but suppose that for our investigation, we want to explicitly generate the automaton that first invokes `WriteRequest` followed by `FSM` where each invocation of the action results in a new state. This is where we bring in the statements that are shown next to a ⋆ in Figure 5. To generate the desired automaton that performs `n + 1` write requests, we overlay the `WriteRequest` and `FSM` with a simple but additional state machine using `reqmode` that updates the `reqmode` state element with every invocation of the actions. The ⋆ lines are added to support this overlaying state machine. However, notice that we use `require` that basically is an assertion that unless the expression evaluates to true, the member cannot be invoked. The loop transitions are removed by introducing the `require` statement because the automaton generation algorithm only explores the respective state when the transition enabling the `require` evaluates to true. This further refines the automaton yielding the corresponding automaton with these additional changes in Figure 7(a). The accepting states are shaded and every transition performs a write request with a value `1`. It is possible to vary the values as well but the concept is easily understood by simplifying the automaton with only one input value. These additions to the semantic model result in the automaton that is desired. Also remember that due to the internal state machine controlled by `mode`, it takes two invocations of `FSM` for a successful write on the `FIFO`.

Executing test cases generated from this automaton raises an exception when the fourth successful write request is made. SpecExplorer's diagnosis indicates that the error occurred when taking the transition from state s_{15} to s_{16}. This suggests that there is a discrepancy between the semantic and the implementation models at the particular state when the FIFO's state is updated to full and there is a write request. SpecExplorer provides an interface for viewing and following the transitions taken in the test cases before the exception was thrown. This is important because it makes it easier for a designer to locate the transition at which a possible error exists. Furthermore, backtracing to the initial state shows the diagnosis or the input sequence that leads to this erroneous state. This is advantageous for the designer because the



(a)



(b)

Figure 7: Automata used for validation of FIFO.

error can also be duplicated. Evidently, the code fragment marked as a possible overflow situation in Figure 6 causes this erroneous behavior. Removing this yields in successful execution of the test cases.

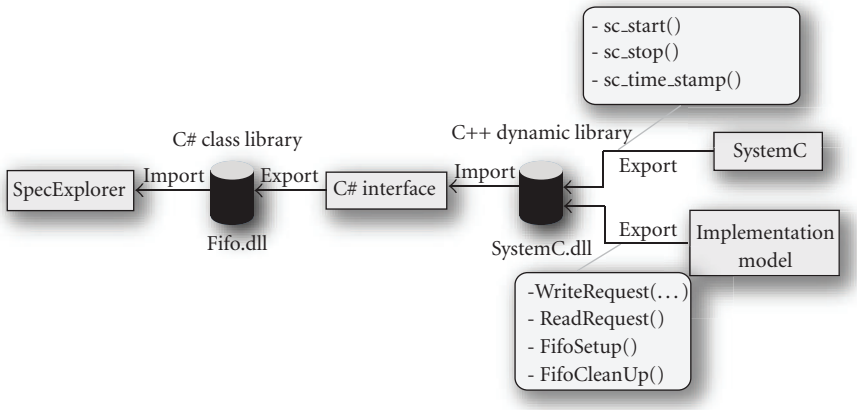
Generating test cases for validating the underflow property is done using a similar approach. The required steps in checking for underflow require the following: (1) adding `ReadRequest` and `FSM` as controllable actions in the exploration settings and (2) making the accepting state to be when the `EMPTY` state variable evaluates to true. The resulting automaton is shown in Figure 7(b).

Executing the test cases for this automaton again show that there is a violation in the implementation model. This occurs at the first successful `ReadRequest` on the transition from state r_3 to state r_4. This is expected because the `EMPTY` state is updated to realize that the FIFO is empty and then the first read request is serviced, but the FIFO has no elements stored in it. Since our implementation model shown in Figure 6 has the error marked as a possible underflow, the implementation throws an exception at this first successful read request. Once again, by uncommentingthe erroneous fragments, the test cases all succeed.

5.1.3. *Wrapper generation*

An integral part to the design flow is the construction of wrappers between SpecExplorer and SystemC. Figure 8(a) shows the process of creating the wrappers and its associated libraries. We describe the flow for generating wrappers and exporting members from the implementation model such that they can be used in the semantic model.

From the SystemC distribution it is necessary to export the `sc_start`, `sc_stop`, and `sc_time_stamp` members where the first drives the simulation, the second stops the simulation, and the third simply serves for debugging purposes. From the implementation model, the necessary members that drive the simulation in that model must be exported, which for the FIFO example are listed. These two sets of exported members are combined into a dynamic C++ library and a snippet of the C++ code used for exporting and importing it into C# is shown in Figure 8(b). The C# wrapper interface imports the `systemc.dll` library for the specific exported members. These are then used in the interface definition in C#. For example in Figure 8(b), we show the definition of the `FSM` member in C#. Note that the class that



(a)

Example of exporting SystemC function

```
extern "C" _declspec(dllexport)          C++
            int hdp_sc_start(int duration);
int hdp_sc_start(int duration) {
            return sc_start(duration);
}
```

Example of importing C# abstract member

```
[DllImport(@"C:\validationASM\systemc.dll"  C#
          ExactSpelling = false)]
public static extern
          int hdp_sc_start(int duration);

 public abstract class SystemCFifoBlock    {
    public static void FSM()      {
       if (hdp_sc_start(1) != 0)      {
          // handle error
       }
    }
 };
```

(b)

Figure 8: Wrapper construction and export code snippets.

this member is contained in is an abstract class and that the types of the members must match those in the semantic model. This C# interface is then compiled as a class library that is called a reference in SpecExplorer. This reference is loaded into SpecExplorer so that test action bindings can be made.

5.2. GCD

The GCD example consists of two input generators and a computation unit. The input generators provide input for the GCD computation and keep changing the inputs. In our case, the input generators simply write an integer value. The GCD computation unit follows Euclid's algorithm. Figure 9 describes the AsmL specification that uses the DE simulation semantics also implemented in AsmL. The components described by `generator1` and `generator2` are simply writing to a channel `arg1` and `arg2`, respectively. The real computation happens in the `computeGCD` member of the `gcd` component. We have a corresponding implementation model for the GCD that is not shown here, but it is available at our website [17].

5.2.1. Exploration and test-case generation

The exploration techniques used for generating the automaton for the GCD example are similar to that explained in the FIFO example. Therefore, without much discussion on the exploration details, we list the two properties that we are

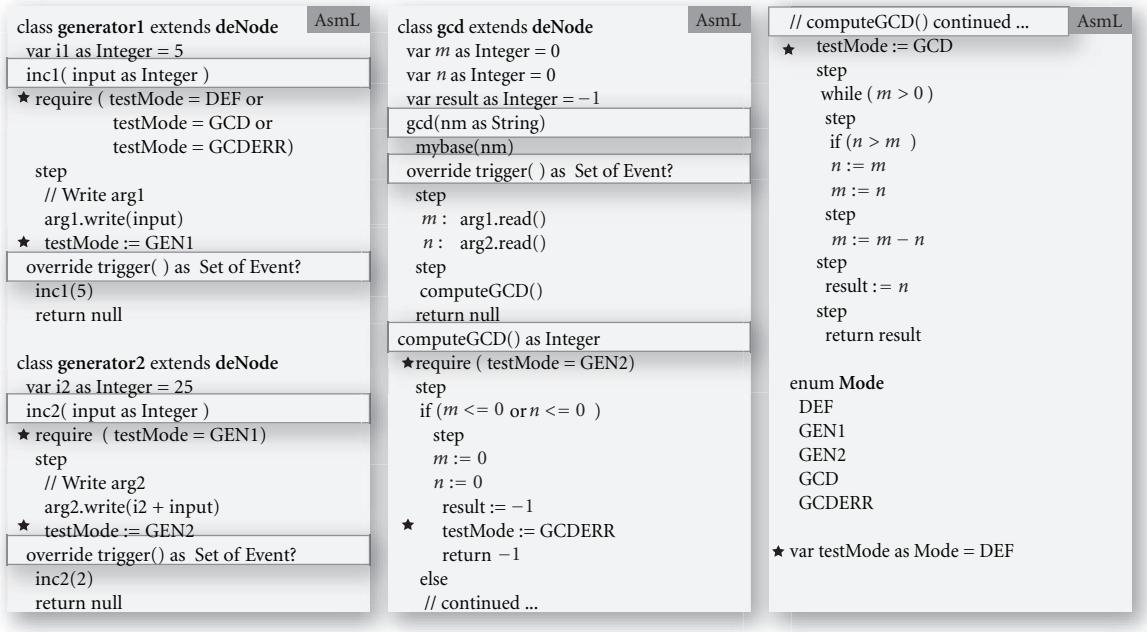


Figure 9: GCD AsmL specification.

interested in validating. They are as follows:

(i) the intended operation of the GCD computation component when encountering invalid input data;

(ii) simply validating when the input sequence consists of either 0 and 1.

To facilitate the exploration for validating the above two properties, we overlay a state machine described by the enumerated type `Mode` standing for request mode and the variable `testMode`. This overlaying procedure is once again similar to that of the FIFO example. Note that in the `computeGCD` member, the result is assigned a -1 value when there is an erroneous computation request. For this to occur, either of the inputs must be zero or any nonnegative integer. So, to validate the first property we do the following: (1) add `inc1`, `inc2`, and `computeGCD` as controllable actions, (2) set `testMode` = `GCDERR` as the accepting states condition, and (3) change the default integer values for the parameter for `inc1` and `inc2` to -1. The resulting automaton is shown in Figure 11.

The transition from state s_2 to s_3 is `computeGCD()/ − 1`, which is in the form of `action/result`. This means that when the `computeGCD` action was invoked, the result of that member was -1 and this should match with the implementation model.

The validation of the second property requires slight alteration to the exploration configuration. The default values for the integers that are used as parameters to `inc1` and `inc2`, and the accepting states condition are altered. We change the default values to allow for `0` and `1` and set the accepting states expression to `testMode` = `GCD` or `testMode` = `GCDERR`. The accepting states are all states where there is a computation that results either in an erroneous or correct computation. The automaton from this exploration configuration is relatively large in size and shown in Figure 10.

5.3. FIR

The finite impulse response semantic model is based on the FIR example from the SystemC distribution. The FIR has a stimulus, computation, and display component. An important property that must hold in order for the FIR to correctly operate is to initiate the reset for a specified number of cycles. The entire FIR requires four clock cycles for the reset to propagate. This is an interesting example because the validation requires exploration techniques that are different from the FIFO and the GCD example. We present snippets of the AsmL specification for the stimulus component used in validating this property in Figure 12(a).

The two members of importance are the `tick` and `stimulusFIR` that increment the cycle counter `counter` and compute whether the reset has completed `reset`, respectively. These are the members that are added as actions in the exploration configuration. So, our exploration settings have (1) `tick` and `stimulusFIR` as controllable actions and (2) `counter` and `reset` as a representative examples of a state grouping. However, note that we introduce a new constrained type `SmallInt` for the `counter` variable. This constrained type is a subset of the integer type with the possible values it can take from `0` to `10`. This cycle `counter` is incremented in `tick` and used for checking the cycle count in `stimulusFIR`. In our semantic model, we had used the DE simulator's internal clock for comparing the cycle count, but we alter this for exploration purposes.

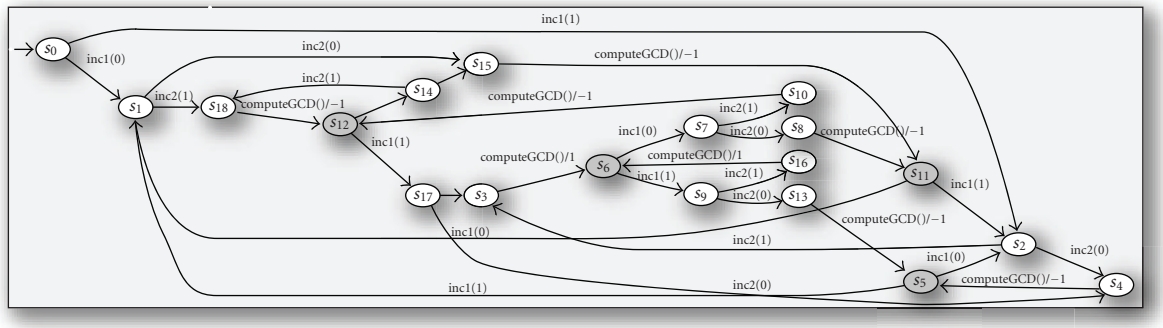


Figure 10: Automaton for validating GCD with 0 or 1 as inputs.

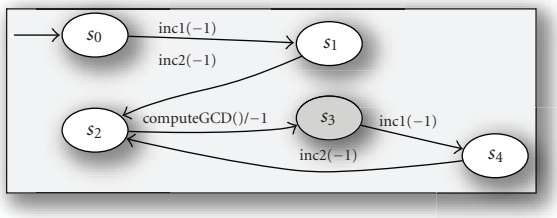


Figure 11: Automaton for validating invalid inputs.

This is because simClock's state is updated within a member of the simulator, mainly `proceedTime`. Therefore, for the automaton to reflect the change in state of `simClock`, additional unnecessary simulation members have to be added and to avoid this, we simply replace it with the `counter` variable and add `tick` to increment the cycle counter.

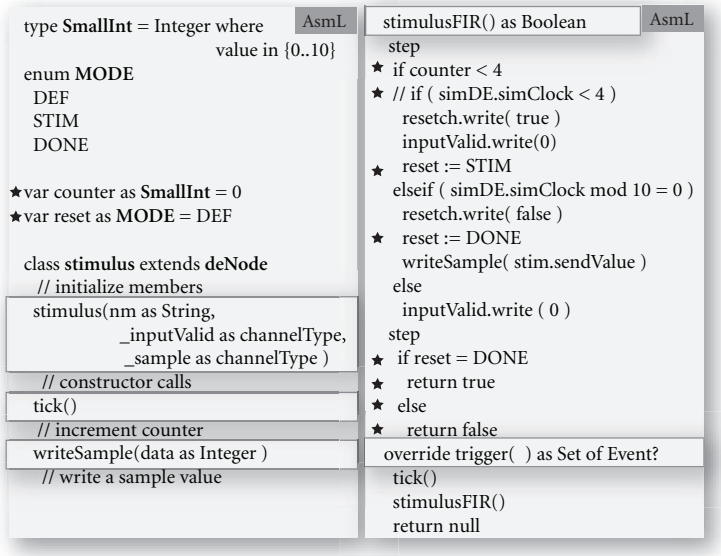
The automaton generated from the above exploration configuration is shown in Figure 12(b). Executing the test cases generated for this automaton show that the reset is indeed held high for four cycles thus showing conformance between the semantic and implementation models. The full source for the semantic model, implementation model, and the wrappers are available at [17] along with the other examples.

6. OUR EXPERIENCE

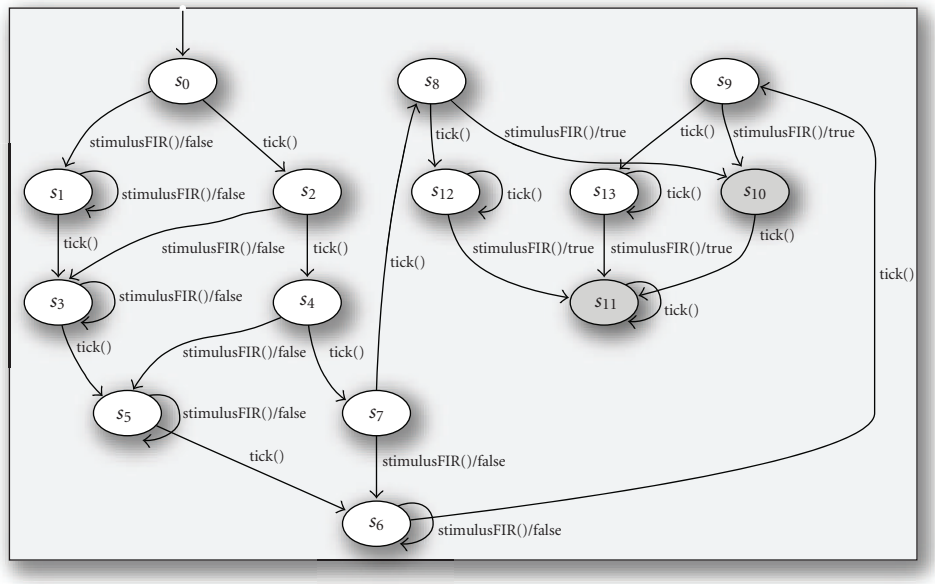
The first phase of semantic modeling is intuitive and simple; even for traditional hardware designers. This is because the additional DE simulation semantics in ASM makes it simple for designers already familiar with SystemC or any other DE simulation environment to describe their semantic models. More importantly, hardware designers do not need to know the complete formal semantics of ASMs, but rather only the same concepts that already exist in traditional hardware description languages such as concurrent statements. The modeling paradigm and extensibility of the DE semantics allows for user-defined channels and modules. The semantics also take into account the nondeterminism mentioned in the SystemC IEEE 1666 standard. This is done by using parallel constructs in AsmL such as the `for all`. However, we do not model all constructs available in SystemC. For example, the `wait` statements are not supported in our version of the AsmL discrete-event semantics. The authors of [12] describe how they incorporate `wait` semantics by instantiating program counters for each thread and saving the counter when switching out. However, any thread-based behavior can be easily modeled as a method-based model [20], and thus refrain from extending our semantics for the sake of clarity. The simulation and debugging of the semantic models are again quite similar to SystemC simulation and debugging with the addition that SpecExplorer allows for step-by-step debugging.

Our experience in creating the implementation models was positive because of the similarity between the semantic and implementation models. In fact, we experienced an easy translation from the semantics to the implementation model. The aspect of simulation and debugging is similar to that at the semantic level.

The most challenging aspect of this methodology was the exploration of the semantic model. This is because, similar to other model-checking engines, the automata generation can suffer from state space explosion. A user can specify state count and transition count constraints in order to time-out the automata generation engine indicating that further tweaking of the semantic model is required. Tweaks to the semantic model are called exploration techniques [4] such as state grouping, state filtering, and parameter selections. These techniques help in abstracting the semantic model to focus on the states and transitions that interest the designer, resulting in a reduction of the state space. Therefore, large designs with a large state space can be incrementally abstracted using these exploration techniques, without having to change much of the semantic model. We acknowledge that the effort in being able to generate the automaton for large designs is more involved, which requires the application of rigorous state space pruning techniques. This however, requires a thorough understanding of how automata are generated from AsmL specifications. Our experience suggests that after exploring on a few examples, the techniques and capabilities of SpecExplorer can be understood clearly; thus making the exploration



(a)



(b)

Figure 12: FIR example.

much simpler for a designer. After successfully obtaining an automaton, the test sequence generation is quick and simple. In fact, generating the simulation sequences corresponds to path traversals on the automaton. This step is automated and takes little time when using the existing algorithms for traversal in SpecExplorer.

Currently, the wrappers are generated manually. The information that these wrappers require are the methods that change the states of interest. Since these methods are already specified in SpecExplorer, we can use this information to automatically generate the C# wrapper with the action binding members. We can do the same for exporting the method functions of the implementation models. However, the SystemC library wrapper only needs to be done once and we provide that. The effort in creating the C# and the SystemC wrappers can be significantly reduced when this step is automated.

7. CONCLUSIONS

We present a model-driven methodology for validating systems modeled in SystemC. Our approach uses the formal

semantic foundation of ASMs via AsmL for semantic modeling and simulation. This formal specification captures the requirements of the intended system at a high abstraction level and defines a contract with the implementation model. The formal specification in AsmL helps in serving any necessary proof obligations required for the designs as well. The designer can then follow this specification when creating the implementation model and since ASMs describe state machines, the mapping to SystemC implementation is intuitive and natural. After the wrapper generations, SpecExplorer's test-case generation can be directed to generate test cases for reaching interesting states in the system. A diagnosis is also provided on these test case executions. SpecExplorer has previously been used for proposing verification methodologies for SystemC designs, but not for test-case generation and execution. This is an important addition to the validation of SystemC designs. We also show examples of directing the test-case generation for the hardware FIFO component, GCD, and the FIR. The FIFO example discusses in detail the exploration techniques and wrappers necessary in employing this methodology. Even though we present our methodology as a model-driven validation approach from the semantic model to the implementation model, it is possible to write semantic models from existing designs and then the semantic model can be used for test-case generation purposes. This is the case for the FIR example that we present.

Our overall experience in using this methodology has been positive in evaluating whether the semantic and implementation models conform to each other. There is an associated learning curve in knowing how to use SpecExplorer and its methods for state space pruning and exploration. However, this methodology scales effectively due to the ease in raising the abstraction level using ASMs. This makes it much easier to focus on only the essential aspects of the system. Furthermore, after familiarizing with the techniques of state space pruning and exploration, the task of directing the test-case generation is routine. Our semantic and implementation models and wrappers will be available on our website.

ACKNOWLEDGMENTS

We acknowledge the support received from the NSF CAREER Grant CCR-023794 and the NSF NGS program Grant ACI-0204028.

REFERENCES

[1] OSCI, SystemC and SystemC Verification, http://www.systemc.org/.

[2] F. Bruschi, F. Ferrandi, and D. Sciuto, "A framework for the functional verification of SystemC models," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 667–695, 2005.

[3] D. Kroening and N. Sharygina, "Formal verification of SystemC by automatic hardware/software partitioning," in *Proceedings of the 3rd ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '05)*, pp. 101–110, Verona, Italy, July 2005.

[4] SpecExplorer, http://research.microsoft.com/specexplorer.

[5] Microsoft Research, Spec#, http://research.microsoft.com/specsharp.

[6] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer, Berlin, Germany, 2003.

[7] Y. Gurevich, "Evolving algebras 1993: Lipari guide," in *Specification and Validation Methods*, pp. 9–36, Oxford University Press, New York, NY, USA, 1995.

[8] A. Gawanmeh, A. Habibi, and S. Tahar, "Enabling SystemC verification using abstract state machines," Tech. Rep., Department of Electrical and Computer Engineering, Concordia University, Montreal, Quebec, Canada, 2004.

[9] A. Habibi, H. Moinudeen, and S. Tahar, "Generating finite state machines from SystemC," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, vol. 2, pp. 76–81, Munich, Germany, March 2006.

[10] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 14, no. 1, pp. 57–68, 2006.

[11] H. D. Patel and S. K. Shukla, "Model-driven validation of SystemC designs," in *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC '07)*, pp. 29–34, San Diego, Calif, USA, June 2007.

[12] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl, "The simulation semantics of SystemC," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 64–70, Munich, Germany, March 2001.

[13] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Towards a tool environment for model-based testing with AsmL," in *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES '03)*, A. Petrenko and A. Ulrich, Eds., vol. 2931 of *Lecture Notes in Computer Science*, pp. 264–280, Montreal, Quebec, Canada, October 2003.

[14] Y. Gurevich, B. Rossman, and W. Schulte, "Semantic essence of AsmL," *Theoretical Computer Science*, vol. 343, no. 3, pp. 370–412, 2005.

[15] M. Barnett and W. Schulte, "The ABCs of specification: Asml, behavior, and components," *Informatica*, vol. 25, no. 4, pp. 517–526, 2001.

[16] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: an overview," in *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '04)*, pp. 49–69, Marseille, France, March 2004.

[17] FERMAT, "Formal Engineering Research with Models, Abstractions and Transformations," http://fermat.ece.vt.edu/.

[18] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillman, and M. Veanes, "Model-based testing of object-oriented reactive systems with spec explorer," Tech. Rep. MSR-TR-2005-59, Microsoft Research, Cambridge, UK, 2005.

[19] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating finite state machines from abstract state machines," in *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA '02)*, pp. 112–122, ACM Press, Roma, Italy, July 2002.

[20] S. A. Sharad and S. K. Shukla, *Correctness Preserving Transformations of System Level Models for Efficient Simulation, Formal Methods and Models for System Design: A System Level Perspective*, Kluwer Academic Publishers, Norwell, Mass, USA, 2005.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 371768, 11 pages
doi:10.1155/2008/371768

Research Article

System Level Modelling of RF IC in SystemC-WMS

Simone Orcioni, Mauro Ballicchia, Giorgio Biagetti, Rocco D. d'Aparo, and Massimo Conti

Dipartimento di Elettronica, Intelligenza artificiale e Telecomunicazioni, Università Politecnica delle Marche, 60131 Ancona, Italy

Correspondence should be addressed to Simone Orcioni, s.orcioni@univpm.it

Received 1 October 2007; Revised 18 February 2008; Accepted 12 April 2008

Recommended by Christoph Grimm

This paper proposes a methodology for modelling and simulation of RF systems in SystemC-WMS. Analog RF modules have been described at system level only by using their specifications. A complete Bluetooth transceiver, consisting of digital and analog blocks, has been modelled and simulated using the proposed design methodology. The developed transceiver modules have been connected to the higher levels of the Bluetooth stack described in SystemC, allowing the analysis of the performance of the Bluetooth protocol at all the different layers of the protocol stack.

1. INTRODUCTION

In recent years, an important issue in electronic design has been the integration of complex intelligence into a single silicon integrated circuit. Today, after years of research and development in the field of silicon technology, it is possible to integrate a complex electronic system, equivalent to millions of transistors, into a single silicon chip: a system-on-chip (SoC). The design of a complex SoC is based on efficient modelling methodologies, on the development of seamless tool chains, on the creation and collection of appropriate test suites, and on the availability of libraries with easily reusable interfaces and clear documentation.

The international technology roadmap for semiconductors (ITRS) [1] and the MEDEA+ electronic design automation (EDA) roadmap [2] highlight heterogeneous systems design among the major challenges in the semiconductor business that need innovative EDA solutions. In moving to top-down synthesis paradigms, descriptions of analog mixed signal (AMS) and radio frequency (RF) parts of an SoC will be the major challenge. A global simulation and performance estimation environment needs to be developed, with high flexibility and powerful capabilities. The intellectual property (IP) reusability will be even more important than it is in the digital domain, though much more complicated to implement.

A unified design framework that encompasses all the key phases of an SoC design, including the simulation of mechanical and simplified electrical physical models, the design and simulation of digital discrete time algorithms, and the design, simulation, and optimization of their implementation on DSP and/or FPGA, is not currently available. Such a design environment would require the ability to cosimulate systems of different natures: mechanical, analog electronic, and digital discrete time electronic systems are just a few examples.

SystemC is a consolidated design language and environment, based on C++, used for system level description of SoCs [3]. SystemC supports a wide range of models of computation and is very well suited to the design and refinement of HW/SW systems from system-level down to register-transfer-level (RTL). However, for a broad range of applications, the digital parts of an electronic system closely interact with the analog parts and thus with the continuous-time environment. Due to the complexity of these interactions, and to the dominant role that the analog parts often play in the overall system behavior, it is essential to consider these analog parts within all the stages of the design process of a mixed-signal system.

The extension of SystemC to mixed-signal applications is currently under development. The aim of this extension is the codesign and cosimulation of mechanical parts and of both analog and digital electronic circuits inside a system. To this end, the SystemC-AMS working group inside OSCI is working on the development of a new library, to be integrated in SystemC, for the description of mixed-signal

systems [4–6]. As application examples, systems in which the analog RF part is integrated within the same chip as the digital part, or a digital control of the injection system in the automotive field are presented.

Recently, the SystemC-WMS (wave mixed signal) library has also been presented [7, 8], implementing a simple SystemC extension to allow mixed-signal modelling and simulation using the concept of incident and reflected waves. This extension to SystemC toward mixed-signal systems enables the creation of a high level executable model of both digital and analog subsystems in the same simulation environment, thus permitting a fast evaluation of the performance of the complete system. SystemC-WMS is suitable to be used in a number of different applications, including, for example, the control of electronic and mechanical systems in automotive applications, or the simulation of analog RF and high-level digital blocks in various wireless network technologies, such as Bluetooth or Zigbee.

Some system level designs of heterogeneous systems have been presented in the literature. In [9], Bjørnsen et al. presented a mixed-signal simulation of two pipelined A/D converter architectures and a CMOS camera-on-a-chip. In [10], Al-junaid and Kazmierski proposed a mixed-signal model of a boost DC-DC converter and a PLL-based frequency synthesiser, while in [11] the physical model of an electric guitar and associated effects, for the implementation in a digital circuit, is reported.

Short-range wireless communications are widely used and of great commercial interest. The design of SoC including wireless transceivers is extremely complex due to the fact that design specifications of the analog and the digital parts are strictly correlated. In [12], an architecture of a ZigBee transceiver has been designed and simulated using the Agilent ADS Ptolemy simulator.

This work presents a system level design of a Bluetooth transceiver, consisting of digital and analog blocks. In Section 2, the SystemC-WMS design framework is presented. Section 3 reports the system level modelling methodology used to design RF mixed signal transceivers starting from the specifications. Section 4 shows an application example regarding the design of a Bluetooth transceiver and the simulation results of the complete system, while the conclusions are reported in Section 5.

2. ANALOG MODULE REPRESENTATION IN SYSTEMC-WMS

SystemC-WMS is a C++ class library, developed to work in conjunction with SystemC, that allows the user to model, simulate, and debug complex systems described at system level, comprising analog and mixed signal blocks, or even blocks operating on different physical domains (e.g., transducers, MEMS, mechanical actuators and systems, etc.). This was achieved by fully exploiting the computation model of SystemC, which is basically an event-driven simulation kernel, by providing a methodology to describe the functionality of analog or mixed signal modules and a flexible system to describe the interconnections between them.

Figure 1: Port symbols using electrical and wave quantities.

In fact, the SystemC kernel's primary function is to schedule the execution of concurrent processes, which describe the functionality of individual modules, according to the interconnections between the modules themselves. The application of this simulation paradigm to an analog system required the specification of an interface that could be used to model the inherent coupling that arises from the interaction of interconnected analog blocks.

As will be discussed shortly, the proposed interface is also able to take care of the dependencies between the modules and the channels directly connected to them, without the need of dealing with the global system topology, so that the simulation can still be carried out by the same event-driven scheduler.

2.1. Module representation based on wave exchanges

Without loss of generality, we can fix our attention to an N-port in the electrical domain, described by its port quantities v_j and i_j, with $j = 1, \ldots, N$. Although these quantities suffice to describe the relations between the module and the external environment, they are not well suited to be part of an event-driven interface because there is no generic way to associate a cause/effect relationship to them.

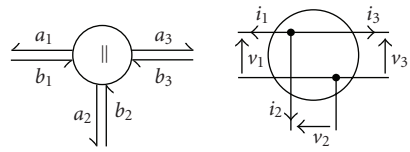
Moreover, in the frequency domain it is customary to use representations of blocks by means of their scattering matrix [13], and the same approach can straightforwardly be extended to other domains as well. For instance, since SystemC is a time-domain simulator, the following definition of incident (a_j) and reflected (b_j) wave:

$$\begin{aligned} a_j(t) &= \frac{1}{2}\left(\frac{v_j(t)}{\sqrt{R_j} + i_j(t)\sqrt{R_j}}\right), \\ b_j(t) &= \frac{1}{2}\left(\frac{v_j(t)}{\sqrt{R_j} - i_j(t)\sqrt{R_j}}\right) \end{aligned} \tag{1}$$

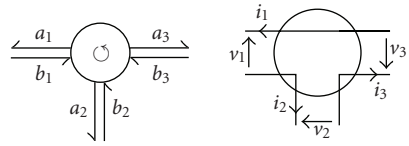
can be used, so that $a_j^2(t) - b_j^2(t)$ is the instantaneous power entering port j, and R_j is a normalisation resistance, that can be assumed to be alike to the characteristic impedance of the transmission line connected to the port if we were working in the frequency domain.

This representation lends itself to the very simple and widespread interpretation of a_j as the cause and of b_j as the effect, so that analog modules thus modelled have well-defined inputs and outputs, as sketched in Figure 1 for a single port. Their activation can thus follow the normal SystemC scheduling practice of re-evaluating those blocks whose inputs have changed.

Having defined the port interface, the description of the dynamics of each analog module can be done by means of a



(a) Parallel interconnection



(b) Series interconnection

Figure 2: Wavechannel symbols corresponding to different port interconnections.

system of nonlinear ordinary differential equations (ODEs) of the following type:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{a}), \\ \mathbf{b} &= \mathbf{g}(\mathbf{x}, \mathbf{a}), \end{aligned} \tag{2}$$

where $\mathbf{f}$ and $\mathbf{g}$ are vector expressions describing the system dynamics, $\mathbf{x}$ is the state vector, while $\mathbf{a}$ and $\mathbf{b}$ are input and output vectors expressed in terms of incident and reflected waves, respectively. These equations will then be solved by an embedded local ODE solver.

2.2. Wavechannels

The use of the incident/reflected wave model greatly simplifies the problem of taking into account the connection between modules, since it can be mandated that modules use incident waves as inputs and produce reflected waves as outputs. This immediately solves the problem of cascading modules, whereas the parallel or series connection can be accomplished by using a new primitive channel that dispatches waves to the modules it connects together, and permits the formulation of a generic and standard analog interface usable across a variety of domains [7].

Such channel behaves similarly to the scattering junction of WDFs [14], which are digital models of analog filters, obtained by the discretisation of individual circuit components. Our approach uses, like in the WDF theory, the *ab* parameters as input/output signals and implements the duties of the scattering junction in a new entity called wavechannel, complying with SystemC conventions for channels.

Wavechannels are the means by which modules described by wave quantities communicate. They can be thought of as a bunch of transmission lines connecting ports to a junction box, in which the lines are tied together, and their role is to model the scattering of waves occurring at the junction. In the current implementation of wavechannels the propagation delay can be excluded, so that their connection to instantaneous blocks may result in the production of delay-free loops.

This fact is dealt with by the standard SystemC delta cycle mechanism which, without further intervention, would just use a fixed-point algorithm to search for the solution of the instantaneous loops, provided that the embedded ODE solver does not advance its state while iterating to find the fixed point.

These delay-free loops that arise from the interconnection of modules are equivalent to algebraic relationships, either between state variables or input/output quantities, giving rise to systems that are globally described by differential algebraic equations (DAEs). It may be worth noticing that, with the fixed-point method, state variable values are not altered during delta cycles, so that only index-1 DAE systems can be solved, i.e., only those admitting a solution of the algebraic part for every value of the state variables.

Let us consider a junction between N ports, each with its own normalisation resistance R_j, and let $\mathbf{v}$ and $\mathbf{i}$ be the voltage and current vectors, respectively, with

$$\begin{aligned} \mathbf{A}_v \mathbf{v} &= 0, \\ \mathbf{A}_i \mathbf{i} &= 0 \end{aligned} \tag{3}$$

being a complete and minimal set of Kirchhoff's equations describing the junction (with $[\mathbf{A}_v]_{ij}, [\mathbf{A}_i]_{ij} \in \{0, \pm 1\}$). We maintain that letting

$$\mathbf{A}_x = \mathbf{A}_v \operatorname*{diag}_{k=1,\ldots,N} R_k, \qquad \mathbf{A}_y = \mathbf{A}_i \operatorname*{diag}_{k=1,\ldots,N} \frac{1}{R_k} \tag{4}$$

the scattering matrix $\mathbf{S}$ (such that $\mathbf{a} = \mathbf{S}\mathbf{b}$) can be computed from (1), (4), and (3), resulting in:

$$S = \begin{bmatrix} \mathbf{A}_x \\ \mathbf{A}_y \end{bmatrix}^{-1} \begin{bmatrix} -\mathbf{A}_x \\ \mathbf{A}_y \end{bmatrix}, \tag{5}$$

where $\mathbf{b}$ are the waves reflected by modules and thus entering the junction, and $\mathbf{a}$ are scattered back from the junction to the modules by that means interconnected.

The above formulation can be used for any kind of junction. In the current implementation it is possible for a user to implement connections of an arbitrary type, by means of a wavechannel, simply by specifying the incidence matrix of the connection itself. To help SystemC-WMS users, the most common types of connections, such as parallel and series structures, half-bridges, and bridges, have already been implemented.

For example, from Kirchhoff's laws, a parallel connection is characterised by the equations:

$$\sum_{j=1}^{N} i_j = 0, \quad v_1 = v_2 = \cdots = v_N, \tag{6}$$

whereas for a series wavechannel we have

$$\sum_{j=1}^{N} v_j = 0, \quad i_1 = i_2 = \cdots = i_N \tag{7}$$

with similar equations that can easily be found for any other type of connection. The derivation of the incidence

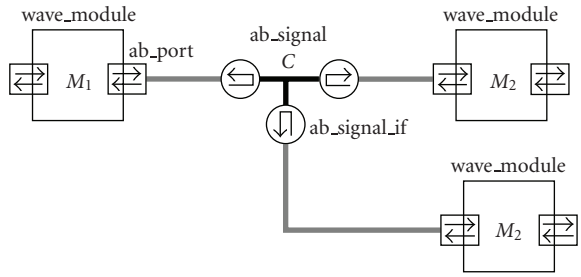


Figure 3: SystemC-WMS class library: sample interconnection structure.

matrix from them is then straightforward. The symbols and the electrical schematic of these wavechannels are shown in Figure 2.

It may be worth noticing here that, if $N = 1$, (6) and (7) simply imply $a_1 = \pm b_1$, and the two channel types are thus able to model the total reflection that takes place at an open circuit or at a shunt, respectively.

2.3. SystemC-WMS class library

To aid the implementation of systems by means of analog blocks described with wave quantities, a number of templates and classes have been designed, as shown in Figure 3: a new kind of port to let modules communicate via wave quantities (*ab_port*), the wavechannel that can interconnect them and that does the real computation of the scattering that occurs at junctions (*ab_signal*), and a template base class (*wave_module*) that eases the declaration of modules by taking care of handling sensitivity lists and port declarations. Moreover, a generic base class *analog_module* (not shown) can be used by noninstantaneous modules to include and access their local ODE solver.

Ports expose an interface that allows users to read (*read*) the incident wave value and to report (*write*) the reflected wave value. Of course, in the same *wave_module*, they can freely be mixed with standard SystemC ports and, in the same design, instantaneous analog modules, signal flow graph (SFG) analog modules (deriving from *analog_module*), and wave modules (deriving from *wave_module*) can be used and mixed together.

An example of declaration of a noninstantaneous wave module is given in Algorithm 1, where the template parameters denote the number of wave ports and the nature of them, with the nature specifying the underlying type of the wave variables together with their name and measurement units for output documentation purposes. After that the user only needs to implement the functions (2) defining the module behavior, as sketched in Algorithm 2.

All the rest is taken care of by the *ab_signal* the module connects to. It is a primitive channel that exploits the two-phase (evaluate/update) scheduling paradigm of SystemC, where the *write(...)* call, occurring during the evaluate phase, determines which channels are to be updated. Then during the update phase, the scattering occurring at the junction is computed, and a new delta cycle triggered if a change

```
struct example:wave_module⟨1, electrical⟩, analog_module
{
  // state variable x is inherited from analog_module
  void field (double *var) const;
  void calculus ();
  SC_CTOR (example) : analog_modulee (· · · )
  {
      SC_THREAD (calculus);
      sensitive << activation;
  }
};
```

Algorithm 1: Typical declaration of a wave module in SystemC-WMS.

```
void example: :field (double *var) const
{
  double a = port−>read ();
  var[0] = f(x, a);                    // evaluate state change
}

void example: :calculus ()
{
  x = 0;                               // state initialization here
  while (step())                       // perform an ODE solver step
   double a = port−>read ();  // read incident wave here
   double b = g(x, a);            //compute reflected wave
   port−>write (b);                    // and send it out
  }
}
```

Algorithm 2: The structure of a typical wave module implementation.

exceeding a certain threshold is detected on any of the scattered waves. Only when no further delta cycles are scheduled, and the core SystemC scheduler advances the time are the embedded local ODE solvers (invoked by the *step()* call) permitted to update their state.

To ease the implementation of complex systems with SystemC-WMS, a number of predefined templates and classes for linear and nonlinear modules have also been provided. The full source code of the SystemC-WMS library and associated device library is freely available under an open source license from the authors' website [15].

3. MODELLING OF ANALOG RF MODULES IN SYSTEMC-WMS

The analog module representation based on (2), which has been used to model different heterogeneous analog systems, ranging from a library of simple linear systems to electrical switching power supplies [7, 8, 15], can easily be extended to better suit electrical circuit requirements in the RF domain. In particular, the representation can be modified by explicitly adding a stochastic signal, which represents the

Table 1: Typical RF module specifications.

Input matching	$s_{11} < \widetilde{s}_{11}$
Output matching	$s_{22} < \widetilde{s}_{22}$
Gain	$s_{21} \simeq \widetilde{s}_{21}$
Isolation	$s_{12} < \widetilde{s}_{12}$
Nonlinearity	$\mathrm{IIP3}_{\mathrm{dBm}} < \widetilde{\mathrm{IIP3}}_{\mathrm{dBm}}$
Noise	$\mathrm{NF} < \widetilde{\mathrm{NF}}$

noise produced by the module itself or by some external interference,

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{a}, \mathbf{n}), \\ \mathbf{b} &= \mathbf{g}(\mathbf{x}, \mathbf{a}). \end{aligned} \tag{8}$$

This is now a stochastic differential equation system, that can be solved in the usual way by considering a realization of the stochastic signal $\mathbf{n}(t)$. If the system can be assumed ergodic, all of the signal features in the probability space can be estimated from time-domain solutions.

To simplify the system-level simulation of (8), some modules can be considered to be instantaneous, resulting in the following simplified representation:

$$\begin{aligned} \mathbf{y}(t) &= \mathbf{S}(\mathbf{a}(t) + \mathbf{n}(t)), \\ \mathbf{b}(t) &= \mathbf{g}(\mathbf{y}(t)), \end{aligned} \tag{9}$$

in the following, for the sake of simplicity, the explicit dependence on time will be neglected. The rationale for (9) is to simulate at system level analog RF modules, like a low noise amplifier (LNA), a mixer, or a power amplifier (PA), in the worse conditions, which happen when considering the S-parameters of the two-port to be constant in frequency, and equal to the specifications. This type of representation allows a straightforward extraction of the parameters of (9) from common RF specifications, such as those shown in Table 1, without the need of assuming any particular circuit implementation for the modules themselves.

As an example to demonstrate how the parameters of (9) can be extracted, and this model applied to specific RF circuits, amplifier and mixer modules will be analyzed in detail in the following.

3.1. Amplifier

Since the nonlinearity involving electrical quantities at the input port can be neglected for amplifiers—indeed there are usually no specifications regarding the nonlinear behavior of the input impedance—(9) can be customised as

$$\begin{aligned} y_1 &= \widetilde{s}_{11}(a_1 + n_1) + \widetilde{s}_{12}(a_2 + n_2), \\ y_2 &= \widetilde{s}_{21}(a_1 + n_1) + \widetilde{s}_{21}(a_2 + n_2), \\ b_1 &= y_1, \\ b_2 &= g_2(y_2). \end{aligned} \tag{10}$$

Figure 4 shows a schematic representation of these relationships.

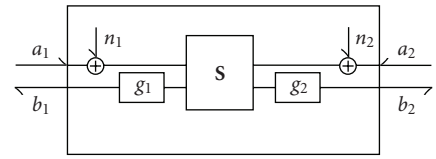


Figure 4: An RF module in SystemC-WMS.

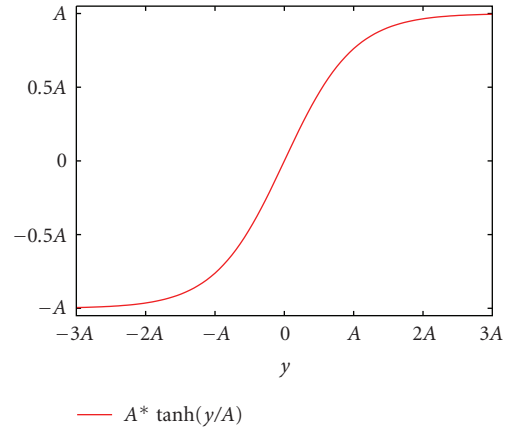


Figure 5: LNA output nonlinearity.

A nonlinear function that well represents the amplifier output nonlinearity, and that can easily be characterized from the specifications reported in Table 1, is the hyperbolic tangent. In the following, we will thus assume that

$$g(y) = A \cdot \tanh\left(\frac{y}{A}\right), \tag{11}$$

so that the small-signal gain is directly set by the values of the linear S-parameters, with a saturation level depending on the value of the parameter A, as can be seen in Figure 5.

3.1.1. Noise

A noisy two-port is generally modelled by adding a couple of noise generators to the input or to the output of a noise-free two-port. A convenient model for our goal is the noise representation based on the a, b parameters, as reported in [16] with output-referred noise generators, or more conveniently with input-referred noise generators, which results in

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = [\mathbf{S}] \begin{bmatrix} a_1 + n_1 \\ a_2 + n_2 \end{bmatrix}, \tag{12}$$

which, when the nonlinearity can be neglected, as it is usually done in the case of noise analysis, corresponds to (10).

A noisy two-port can be characterized, in terms of noise performance, by its noise factor F defined as [16]

$$F = \frac{(S/N)_{\mathrm{I}}}{(S/N)_{\mathrm{O}}} = 1 + \left(\frac{N_{\mathrm{AI}}}{N_{\mathrm{I}}}\right), \tag{13}$$

where N_{AI} is the added noise power, referred to the input, and N_{I} is the noise power entering the input port. The noise

factor, when expressed in decibel, is called noise figure, $NF = 10\log(F)$.

Since the noise specification of an amplifier is usually expressed only in terms of a scalar variable (F or NF), only the parameters of one of the two noise generators can be determined. Because of this we have chosen to put $n_2 = 0$ in (12).

A noisy resistor can give a maximum noise power equal to

$$P_{\mathrm{MAX}} = \frac{\overline{v_n^2}}{4R} = \frac{4kTRB}{4R} = kTB, \tag{14}$$

where k is the Boltzmann constant, T is the temperature, and B the bandwidth. This relationship can be applied to (13) to obtain

$$\sigma_n^2 = \overline{n_1^2} = kTB(F-1). \tag{15}$$

This relationship lets us add a stationary white noise wave generator of variance that depends only upon the specifications reported in Table 1.

3.1.2. Nonlinearity

The nonlinearity performance analysis of an amplifier module can be performed, following [17], by approximating the nonlinearity (11) by its Taylor expansion truncated to the third order:

$$g(y) = y - \frac{y^3}{3A^2} = \alpha_1 y + \alpha_3 y^3, \tag{16}$$

which leads to the following expression for the OIP3 :

$$\widehat{\mathrm{OIP3}} = \alpha_1\sqrt{\frac{4}{3}\left|\frac{\alpha_1}{\alpha_3}\right|} = 2A. \tag{17}$$

We have used the hat sign in the OIP3 to underline the fact that (17) regards the amplitude of the signals, which in a, b parameters is expressed in $\sqrt{W}$. Since the IIP3 or OIP3 are usually expressed in terms of power, (17) must be translated from $\sqrt{W}$ to W. It is straightforward to relate the amplitude of incident and reflected waves to the mean power available at a port,

$$\begin{aligned} P_{\mathrm{av}} &= \frac{1}{T}\int \left(b^2(t) - a^2(t)\right)\mathrm{d}t \\ &= \frac{1}{T}\int b^2(t)\mathrm{d}t = \frac{\hat{b}^2}{2} \end{aligned} \tag{18}$$

obtained for sinusoidal signals and neglecting the incident wave at the port. This relation can be rewritten as $\hat{b} = \sqrt{2P_{\mathrm{av}}}$, or $\widehat{\mathrm{OIP3}} = \sqrt{2\mathrm{OIP3}}$. Since the specifications are usually expressed in terms of $\mathrm{IIP3}_{\mathrm{dBm}}$, it is easy to find that

$$\mathrm{OIP3}_{\mathrm{dBm}} = \mathrm{IIP3}_{\mathrm{dBm}} + \mathrm{TPG}_{\mathrm{dB}}. \tag{19}$$

Finally, from the preceding equations, we can express the nonlinear parameter characterizing (10), in term of specifications as $A = 1/2\widehat{\mathrm{OIP3}}$, or

$$A = \frac{1}{2}\sqrt{2\cdot\mathrm{OIP3}}. \tag{20}$$

Table 2: Amplifiers' specifications.

Parameter	LNA spec.	PA spec.	PGA spec.
$\widetilde{s}_{11}$	−15 dB	−15 dB	—
$\widetilde{s}_{12}$	−40 dB	−40 dB	—
$\widetilde{s}_{21}$	18 dB	18 dB	—
$\widetilde{s}_{22}$	−15 dB	−15 dB	—
$\widetilde{\mathrm{IIP3}}$	0 dBm	18 dBm	—
$\widetilde{\mathrm{NF}}$	4 dB	—	34 dB

Equation (10), which can be assumed to be a model valid for every type of amplifier, can finally be rewritten with all previously extracted parameters as

$$\begin{aligned} b_1 &= \widetilde{s}_{11}(a_1 + n_1) + \widetilde{s}_{12}a_2, \\ y_2 &= \widetilde{s}_{21}(a_1 + n_1) + \widetilde{s}_{21}a_2, \\ A &= \frac{1}{2}\sqrt{2\cdot\widetilde{\mathrm{OIP3}}}, \\ b_2 &= A\cdot\tanh\left(\frac{y_2}{A}\right), \\ \overline{n_1^2} &= kTB(\widetilde{F} - 1). \end{aligned} \tag{21}$$

3.1.3. Implementation

With the facilities provided by SystemC-WMS and the equations previously reported in (21), it is straightforward to implement a high-level module to model an LNA. Its full implementation is reported in Algorithm 3, where *noise* is another library class that generates a sequence of gaussian random numbers, of variance equal to the desired noise power, and then modulates the result into the requested frequency band, so as to produce a brick wall-shaped noise spectrum with the desired properties.

The LNA so defined can then be used in constructs like the one reported in Algorithm 4, where the tags like dB or GHz that appear after several of the numerical constants are simple macros that implement the trivial scaling and log-to-linear conversions of the parameters as needed.

3.2. Mixer

The relations (8) for the mixer take the following form:

$$\begin{aligned} b_1 &= \widetilde{s}_{11}(a_1 + n_1) + \widetilde{s}_{\mathrm{LO1}}a_{\mathrm{LO}} + \widetilde{s}_{12}a_2, \\ y_2 &= K(1 + \widetilde{s}_{22})\left[(1 + \widetilde{s}_{11})(a_1 + n_1) + \widetilde{s}_{\mathrm{LO1}}a_{\mathrm{LO}} + \widetilde{s}_{12}a_2\right] \\ &\quad \times \frac{a_{\mathrm{LO}}}{A_{\mathrm{LO}}} + \widetilde{s}_{22}a_2, \\ A &= \frac{1}{2}\sqrt{2\cdot\widetilde{\mathrm{OIP3}}}, \\ b_2 &= A\cdot\tanh\left(\frac{y_2}{A}\right), \end{aligned} \tag{22}$$

```
struct lna : wave_module ⟨2, electrical⟩
{
  SC_HAS_PROCESS (amplifier_scatter);
  amplifier_scatter (sc_core : : sc_module_name name,
    scatter S11, scatter S12, scatter S21, scatter S22,
    double IIP3, double F, double f0, double band,
    double R01 = 50 ohm, double R02 = 50 ohm
  ) :
    S11(S11), S12(S12), S21(S21), S22(S22),
    A (0.5 ∗ sqrt (2 ∗ IIP3) ∗S21),
    n1 (k ∗ T0 ∗ band ∗ (F − 1), f0, band)
  {
    SC_METHOD (calculus); sensitive << activation;
    // set normalization resistances on ports:
    port [0] <<= R01;
    port [1] <<= R02;
  }
private:
  void calculus ()
  {
    double t = sc_core : : sc_time_stamp (). to_seconds ();
    double a1 = port [0]−>read() + n1(t);
    double a2 = port[1]−>read ();
    double y1 = S11∗a1∗S12∗a2;
    double y2 = S21∗a1∗S22∗a2;
    port [0]−>write (y1);
    port [1]−>write (A∗tanh (y2/A));
  }
private: /* data members */
  const double S11, S12, S21, S22;
  const electrical :: wave_type A;
  noise n1;
};
```

Algorithm 3: Definition of the LNA module in SystemC-WMS.

```
ab_signal ⟨electrical, parallel⟩ antenna(50 ohm);
ab_signal ⟨electrical, parallel⟩ amplified(50 ohm);
source ⟨electrical⟩ src1("RX-SIGNAL", cfg : : wave);
src1(antenna, ...);   // connection of signal source
lna amp1("LNA1,"
  −15 dB, −40 dB, 18 dB, −15 dB,   //S-parameters
  0 dBm, 4 dB,   // nonlinearity and noise specs
  2.485 GHz, 1 MHz);      // band specification
amp1 (antenna, amplified);   //connections of the LNA
```

Algorithm 4: Example of a typical usage of the LNA module.

The mixer has been modelled by a two-port electrical module with an SFG input a_{LO}, that is the input of the local oscillator. This signal has been modelled by a sinusoidal function:

$$a_{LO} = A_{LO} \cos\left(\omega_{LO} t + \phi\right), \tag{23}$$

where $A_{LO} = \sqrt{2P_{avLO}}$ is a function of local oscillator power. The reflected wave at the input port of the mixer b_1 is linearly related to all the inputs of the module by the following RF specifications of the mixer:

Table 3: TX-RX Mixer specifications.

Parameter	TX spec.	RX spec.
$\widetilde{s}_{11}$	−15 dB	−15 dB
$\widetilde{s}_{12}$	−90 dB	−90 dB
$\widetilde{s}_{21}$	2 dB	12 dB
$\widetilde{s}_{22}$	−15 dB	−15 dB
$\widetilde{s}_{LO1}$	−86 dB	−90 dB
$\widetilde{IIP3}$	15 dBm	12 dBm
$\widetilde{NF}$	—	24 dB

(i) the input reflection coefficient $\widetilde{s}_{11}$;

(ii) the power transfer from the output port $\widetilde{s}_{12}$;

(iii) the power transfer from the local oscillator $\widetilde{s}_{LO}$.

The relationship that characterizes y_2 has been obtained considering that the output current of the mixers based on Gilbert multiplier is approximately $i'_2(t) \approx G v_1(t) v_{LO}(t)$. On the base of the proposed model, the normalized current $i'_{2n}(t)$ can be expressed as a function of the incident and reflected wave at the input port as follows:

$$i'_{2n}(t) = G\sqrt{R_{02}R_{01}}\,(a_1(t) + n_1(t) + b_1(t))\,a_{LO}(t). \tag{24}$$

Now, substituting the expression for b_1 in (24), we obtain

$$i'_{2n} = G\sqrt{R_{02}R_{01}}\left[(1+\widetilde{s}_{11})(a_1+n_1) + \widetilde{s}_{LO}a_{LO} + \widetilde{s}_{12}a_2\right]a_{LO}. \tag{25}$$

The output port of the mixer has been, therefore, modelled by a controlled current generator that provides the normalized current of (25) with a parallel load of impedance R_L. The relationship between b_2 and a_2 of such a kind of circuit can be expressed as follows:

$$y_2 = \frac{1+\widetilde{s}_{22}}{2} i'_{2n} + \widetilde{s}_{22}a_2 \tag{26}$$

with $\widetilde{s}_{22} = (R_L - R_{02})/(R_L + R_{02})$. Now substituting (25) in (26), we obtain the final expression for b_2, where the power conversion gain is provided by the factor $K = G\sqrt{R_{02}R_{01}}\,A_{LO}/2$.

4. BLUETOOTH TRANSCEIVER

4.1. Bluetooth RF specifications

Bluetooth [18] is a widely used wireless standard for devices that have regular charge (e.g., mobile phones) and in applications like handsfree audio and file transfer.

The Bluetooth frequency band is comprised within the industrial, scientific, and medical (ISM) band, between 2.4000 GHz and 2.4835 GHz and is subdivided into 79 channels. Three power classes of devices are defined, with

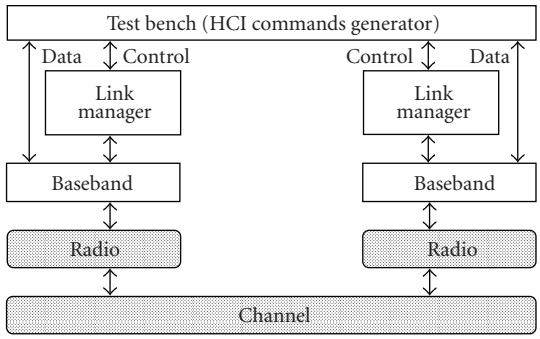


Figure 6: Bluetooth baseband and link layer, as modelled in [19] (in white), with the addition of the radio layer (in gray).

different output power requirements. For a class 2 device, that has a rated maximum operating distance of 10 m, the maximum output power is nominally 1 mW (0 dBm), and must always be within the range of 0.25 mW (−6 dBm) to 2.5 mW (4 dBm). The modulation employs a Gaussian frequency shift keying (GFSK) with BT = 0.5, a modulation index that must lie between 0.28 and 0.35, and a symbol rate of 1 MS/s. The symbol rate accuracy must be better than 20 μs/s, the minimum frequency deviation must never be smaller than 115 kHz. The zero crossing error, that is the time difference between the ideal symbol period and the measured crossing time, must be less than 0.125 microsecond (1/8 of a symbol period at 1 MS/s). The transmitter initial center frequency accuracy must be 75 kHz. The Bluetooth receiver must have an actual sensitivity level of −70 dBm or better, where the actual sensitivity level is defined as the input level for which the BER is 0.001.

4.2. RF layer architecture

The Bluetooth baseband and link layers have previously been modelled in SystemC [19], without considering the analog radio layer. The heterogeneous modelling methodology allowed by SystemC-WMS, described earlier in this paper, permits the modelling and simulation of the lower layers of the Bluetooth stack including the analog radio layer, as reported in Figure 6. The possibility of performing a fast and yet detailed simulation, requiring a CPU time of less than 10 seconds for simulating the transmission, RF propagation, and reception of each bit (i.e., 1 microsecond of simulated time), allows the verification of the effect of changes in the design specifications of the analog blocks on the overall system performance at packet level.

In the rest of this section, a detailed description of the radio layer is reported. Many architectures of transmitters and receivers for the 2.4 GHz band have been proposed, some of them suitable for a single standard, some for multiple standards. The receivers can be classified according to their operation as superheterodyne, image-reject, zero-IF, or low-IF. Analogously, the transmitters can be classified as superheterodyne, direct-up, or two-step-up. A review of the state of the art in wireless transceivers can be found in [20].

The designed transceiver is composed of a transmitter and a receiver whose structures are reported in Figures 7 and 8, respectively. The low-IF receiver (RX) is derived from the designs reported in [21, 22]. The LNA amplifies the signal from the antenna. The voltage controlled oscillator (VCO) generates the in-phase (I) and quadrature-phase (Q) components for the RF to intermediate frequency (IF) down conversion performed by the mixer. A low-band filter centered at the 2 MHz IF rejects the image signal. The programmable gain amplifier (PGA) amplifies the signal to the desired level. The IF to baseband down conversion is performed digitally after the A/D conversion, whose resolution can easily be changed in the code. The specifications of the single blocks, derived from the Bluetooth RF specifications, are reported in Tables 2-3.

The direct-conversion transmitter (TX) is reported in Figure 7, and is derived from the designs reported in [22, 23]. A digital Gaussian filter is implemented using a lookup Table (LUT), the signal is integrated and then the baseband in-phase (I) and quadrature (Q) components are generated. The I and Q digital signals are converted to analog signals, low pass filters clean the signals and single side band mixers upconvert them directly to 2.4 GHz. A power amplifier amplifies the signal before feeding it to the antenna. The specifications for the single blocks are reported in Tables 2-3.

4.3. Simulation results

The complete system with transmitter and receiver was simulated with the specifications previously discussed.

The propagation of the signal from the transmitter to the receiver was modelled by a classical additive white Gaussian noise (AWGN) channel with an attenuation factor of $\lambda^2/(4\pi d^2)$, where d is the distance between transmitter and receiver and λ is the RF wavelength. The noise level introduced by the channel was set to kT_0 with $T_0 = 290$ K, that corresponds to −144 dBm/kHz.

The simulation of the transmission of 50 bits took about 8 minutes on a 3 GHz Pentium D processor. Figure 9 shows the simulation results for different signals sensed inside the digital portions of the transmitter and of the receiver during this simulation. The first signal is the digital data stream to be transmitted, at the input of the Gaussian filter, followed by the output of the gaussian filter itself. The third signal is the received signal at the output of the quadricorrelator followed again by the digital data stream obtained at the output of the receiver.

The performance of the complete system was measured by means of the normalized root mean square error (RMSE), computed as

$$\text{Normalised RMSE} = \frac{\overline{\left(s_{\text{i}}(t) - s_{\text{o}}(t)\right)^2}}{\overline{\left(s_{\text{i}}(t)\right)^2}}, \tag{27}$$

where the transmitted signal s_{i} is sensed at the output of the Gaussian filter, and the received signal s_{o} at the output of the quadricorrelator. Both signals have been synchronized and rescaled before applying (27).

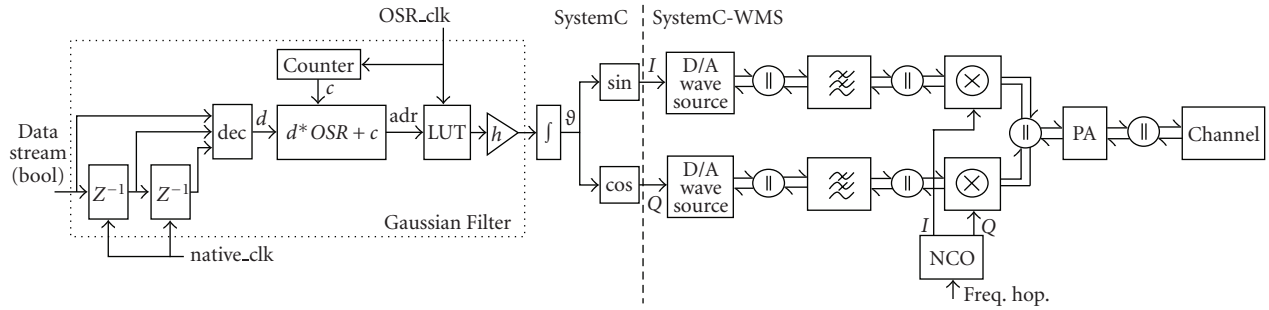


Figure 7: Transmitter block diagram.

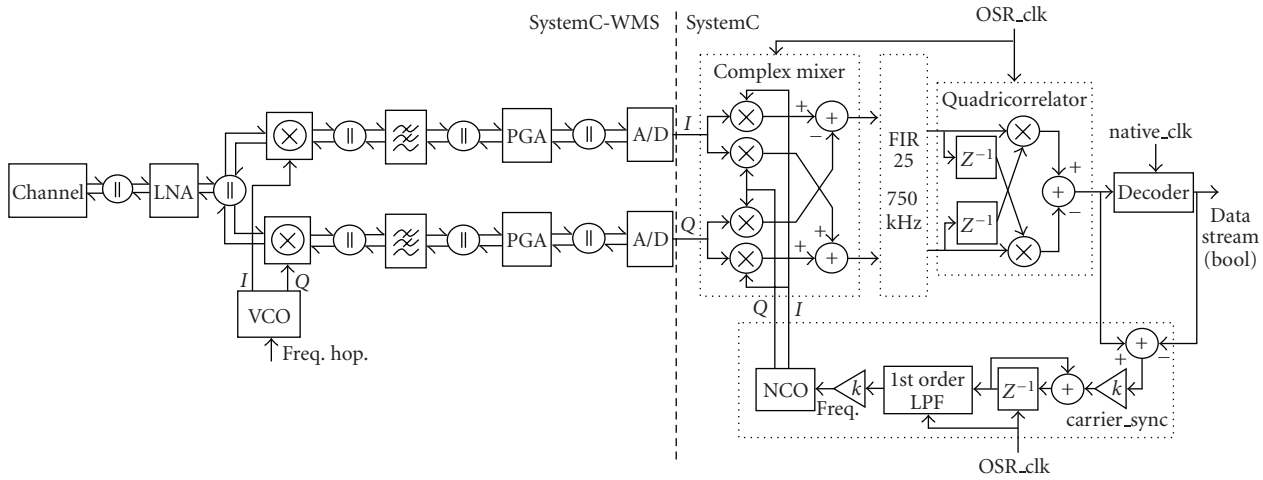


Figure 8: Receiver block diagram.

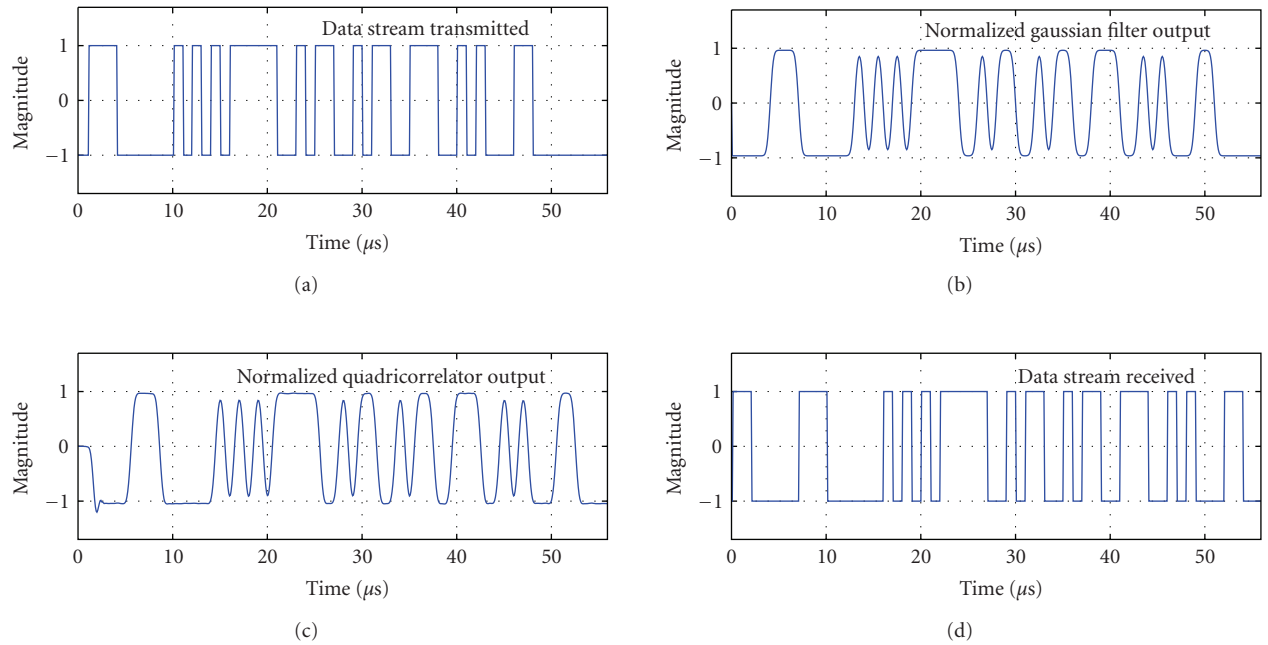


Figure 9: Example of transmitted and received signals inside the digital portions of the transceiver. All the magnitudes have been normalised according to the available resolution.

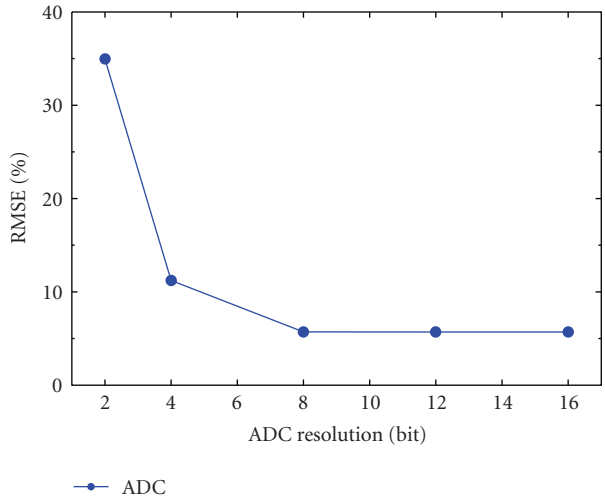


Figure 10: Normalized RMSE of the whole system as a function of ADC resolution.

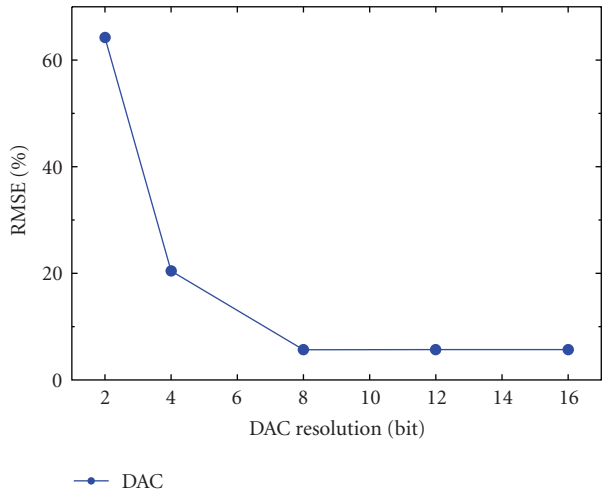


Figure 11: Normalized RMSE of the whole system as a function of DAC resolution.

The RMSE is affected by several factors, related to both the transmitter and the receiver, such as output-stage nonlinearity, DAC nonideality, LPF nonideality, and VCO nonideality.

Figures 10-11 show the simulated RMSE performance versus the ADC and DAC resolutions. The curve in Figure 10 was obtained by calculating the RMSE for different values of the ADC resolution, ranging from 2 to 16, while the DAC resolution was kept fixed at the maximum value. Analogously, the same sweep was performed on the DAC resolution, keeping constant the other, in order to obtain the results shown in Figure 11.

The receiver was designed in order to have a noise figure of about 12 dB [22], as can easily be verified applying Friis equation [16] to the noise figure and power gain

Table 4: Transducer power gain and noise figure of various receiver stages as obtained from simulation results.

Receiver	TPG	NF
LNA	17.7 dB	4 dB
MIXER	14.4 dB	24.2 dB
PGA	12 dB	34.2 dB
RX	41.1 dB	12.9 dB

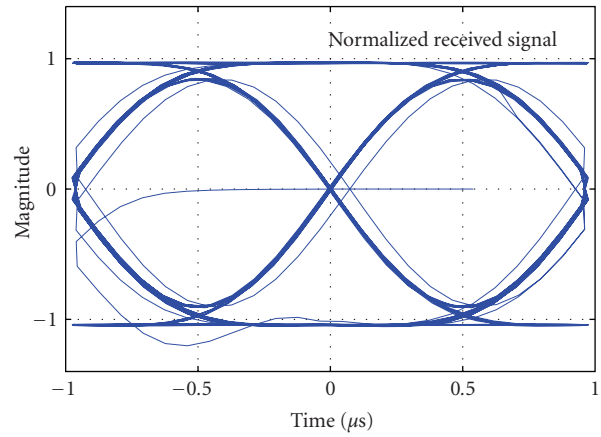


Figure 12: Eye-diagram measured at quadricorrelator output, with only thermal noise in the channel and noisy receiver as per the specifications. The bit time is $T_b = 1\ \mu$s.

specifications given for each of the receiver components in Tables 2-3:

$$\mathrm{NF_{RX}} = 10\log\left(F_{\mathrm{lna}} + \frac{F_{\mathrm{mixer}} - 1}{\mathrm{TPG_{lna}}A_{\mathrm{splitter}}} + \frac{F_{\mathrm{pga}} - 1}{\mathrm{TPG_{lna}}A_{\mathrm{splitter}}\mathrm{TPG_{mixer}}}\right) = 12\,\mathrm{dB}, \tag{28}$$

where A_{splitter} is the power attenuation due to the splitter at the output of the LNA, that is of about 3 dB.

In order to experimentally evaluate the noise figure of the receiver from simulations, two transmissions of the same 50-bit sequence had been simulated, one with noise and another without noise. The power of the signal (S) and of the noise (N) at the input and at the output of each stage of the receiver was then calculated from the data provided by this two simulations, leading to the following result:

$$\mathrm{NF_{RX}} = 10\,\log\left(\frac{S_{\mathrm{I\,lna}}}{N_{\mathrm{I\,lna}}}\cdot\frac{N_{\mathrm{opga}}}{S_{\mathrm{opga}}}\right) = 12.9\,\mathrm{dB}. \tag{29}$$

These simulations allowed us to also evaluate the power gain and the noise figure of each stage of the receiver. These data are reported in Table 4, and it is possible to observe a substantial agreement between them and the corresponding specifications in Tables 2-3.

Figure 12 shows the eye-diagram performance of the receiver, while Figure 13 shows the power spectrum at the receiver input, obtained by the transmission of a sequence

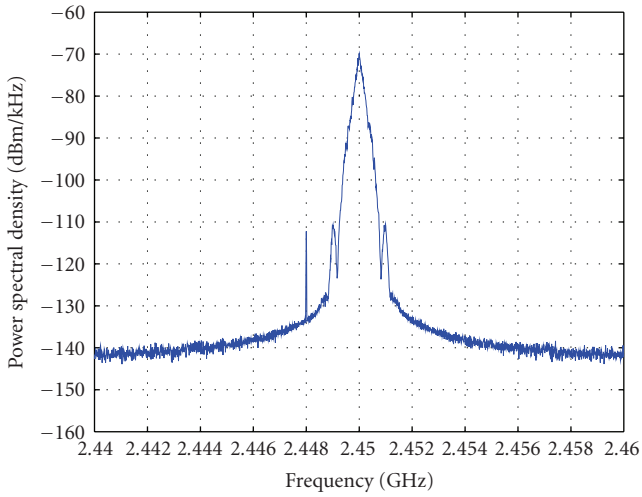


Figure 13: Power spectrum density of the RF signal measured at the LNA input.

of 1024 random bits. The narrow peak at 2.448 GHz is the local oscillator feedback due to the imperfect mixer isolation (−90 dB) and finite reverse gain (−40 dB) of the LNA.

5. CONCLUSIONS

This work proposes an effective methodology that simplifies the modelling of the interaction between analog models belonging to heterogeneous domains, as well as model reuse. By using power waves as standard input/output signals for analog modules, these can be independently modelled and freely interconnected together in arbitrary topologies. The main peculiarity of SystemC-WMS is to allow nonlinear dynamical analog models simulation seamlessly with interconnections that automatically take care of load and interconnection effects.

This paper also proposes a methodology for system level modelling of RF systems in SystemC-WMS, directly based on the high-level specifications of the RF modules. The proposed methodology was applied to the modelling of a Bluetooth transceiver, and the simulation results reported and discussed.

REFERENCES

[1] "ITRS, 2005 Edition: Design," International Technology Roadmap for Semiconductors, December 2005.

[2] "MEDEA Electronic Design Automation (EDA) Roadmap, 5th release," MEDEA+, September 2005.

[3] The Open SystemC Initiative, (OSCI), "SystemC documentation," http://www.systemc.org/.

[4] A. Vachoux, C. Grimm, and K. Einwich, "SystemC-AMS requirements, design objectives and rationale," in *Proceedings of the Design, Automation and Test in Europe (DATE '03)*, pp. 388–393, Munich, Germany, March 2003.

[5] A. Vachoux, C. Grimm, and K. Einwich, "Analog and mixed signal modelling with SystemC-AMS," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '03)*, vol. 3, pp. 914–917, Bangkok, Thailand, May 2003.

[6] OSCI Analog/Mixed-Signal Working Group, (AMSWG), "SystemC-AMS documentation," http://www.systemc.org/apps/group-public.

[7] S. Orcioni, G. Biagetti, and M. Conti, "SystemC-WMS: mixed signal simulation based on wave exchanges," in *Applications of Specification and Design Languages for SoCs*, A. Vachoux, Ed., ChDL, chapter 10, pp. 171–185, Springer, Berlin, Germany, 2006.

[8] S. Orcioni, G. Biagetti, and M. Conti, "SystemC-WMS: a wave mixed signal simulator," in *Proceedings of the Forum on Specifications & Design Languages (FDL '05)*, pp. 61–72, Lausanne, Switzerland, September 2005.

[9] J. Bjørnsen, T. E. Bonnerud, and T. Ytterdal, "Behavioral modeling and simulation of mixed-signal System-on-a-Chip using SystemC," *Analog Integrated Circuits and Signal Processing*, vol. 34, no. 1, pp. 25–38, 2003.

[10] H. Al-Junaid and T. Kazmierski, "Analogue and mixed-signal extension to SystemC," *IEE Proceedings: Circuits, Devices and Systems*, vol. 152, no. 6, pp. 682–690, 2005.

[11] F. Gambini, M. Conti, S. Orcioni, F. Ripa, and M. Caldari, "Physical modelling in SystemC-WMS and real time synthesis of electric guitar effects," in *Proceedings of the 5th Workshop on Intelligent Solutions in Embedded Systems (WISES '07)*, pp. 87–100, Madrid, Spain, June 2007.

[12] N.-J. Oh and S.-G. Lee, "Building a 2.4-GHz radio transceiver using IEEE 802.15.4," *IEEE Circuits and Devices Magazine*, vol. 21, no. 6, pp. 43–51, 2005.

[13] K. Kurokawa, "Power waves and the scattering matrix," *IEEE Transactions on Microwave Theory and Techniques*, vol. 13, no. 2, pp. 194–202, 1965.

[14] A. Fettweis, "Pseudo-passivity, sensitivity, and stability of wave digital filters," *IEEE Transactions on Circuits Theory*, vol. 19, no. 6, pp. 668–673, 1972.

[15] G. Biagetti, M. Conti, and S. Orcioni, "SystemC-WMS home page," http://www.deit.univpm.it/systemc-wms/.

[16] G. D. Vendelin, A. M. Pavio, and U. L. Rohde, *Microwave Circuit Design Using Linear and Nonlinear Techniques*, John Wiley & Sons, New York, NY, USA, 1990.

[17] B. Razavi, *RF Microelectronics*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1998.

[18] "Specification of the Bluetooth System," 1st ed., Bluetooth, December 1999.

[19] M. Conti and D. Moretti, "System level analysis of the Bluetooth standard," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, vol. 3, pp. 118–123, Munich, Germany, March 2005.

[20] P.-I. Mak, S.-P. U, and R. P. Martins, "Transceiver architecture selection: review, state-of-the-art survey and case study," *IEEE Circuits and Systems Magazine*, vol. 7, no. 2, pp. 6–24, 2007.

[21] D.-C. Chang and T.-H. Shiu, "Digital GFSK carrier synchronization," in *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems (APCCAS '06)*, pp. 1523–1526, Singapore, December 2006.

[22] H. Darabi, S. Khorram, H.-M. Chien, et al., "A 2.4-GHz CMOS transceiver for Bluetooth," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 12, pp. 2016–2024, 2001.

[23] T. A. D. Riley and M. A. Copeland, "A simplified continuous phase modulator technique," *IEEE Transactions on Circuits and Systems II*, vol. 41, no. 5, pp. 321–328, 1994.