Section III

Micro-architecture design

12

SystemC Models and Annotations

12.1 Introduction **12**-1

12.2 Design Methodology **12**-3
Abstraction Levels • System Modeling Methodology • Design Space Exploration

12.3 Annotation of System-Level Models **12**-7
System-Level Modeling Objects • Time Annotation • Power Intent • Security Annotation

12.4 SystemC Models - Back-Annotation **12**-12
Abstract Processor Models • Hierarchical FSMs in SystemC • Asynchronous Inter-Module Communication in SystemC • SystemC Security Annotation of a Watchdog Timer • Behavioral PLL Model in SystemC-AMS

12.5 Automated Monitoring **12**-28
Statistical Classes for System-Level Models • Advanced System Monitoring

12.6 Open Issues **12**-31

Miltos D. Grammatikakis
Hellenic Mediterranean University
Heraklion, Greece

George Kornaros
Hellenic Mediterranean University
Heraklion, Greece

Li Wenzhen
Self-Employed

Brent Sherman
Intel Corporation
Hillsboro, OR
USA

12.1 Introduction

The number of transistors on a chip grows exponentially, pushing technology toward highly integrated systems-on-a-chip (SoC) [1]. Existing design tools fail to exploit all the possibilities this technological leap offers, and shorter-than-ever time-to-market trends drive the need for innovative design methodology and tools. It is expected that a product quality and productivity leap can only be achieved by focusing on higher levels of abstraction, enabling optimization of the top part of the design where important algorithmic and architectural decisions are made, and massive reuse of predesigned system and block components occurs. Failure to achieve the required productivity leap would result in missed market opportunities and escalating design costs.

Detailed VHDL or Verilog models are inadequate for system-level description due to poor simulation performance. Advanced system-level modeling may lead from a high-level system model derived from initial specifications through successive functional decomposition and refinement to implementing an optimized, functionally correct, unambiguous protocol without hazards such as deadlocks, livelocks, or race conditions.

A popular open-source C++ system-level design and simulation library that supports clock-accurate (also called cycle-accurate) modeling is SystemC [2] [3]. SystemC was initially standardized by OSCI in 1999, and SystemC 1.0 was released in the year 2000. SystemC 2.0 introduced transaction-level modeling in 2001. The language reference has been ratified as IEEE Std 1666-2005, with a subsequent version in 2011 (compatible with C++11), and more recently in 2023 (compatible with C++17). Currently, a consortium of EDA/semiconductor companies lead the Accellera Systems Initiative to coordinate development and plan future IEEE standardization efforts.

SystemC consists of a collection of C++ classes describing mainly hardware architecture concepts and a simulation kernel implementing the runtime semantics. The classes model basic concepts used by HDLs, such as fixed-point data types, modules, ports, signals, time, and abstract concepts such as interfaces, communication channels, and events. SystemC 3.0+ allows the development and exchange of fast system models via IP-XACT (IEEE Std 1685), providing seamless integration with tools from a variety of vendors [4]. Accellera provides an open-source sequential SystemC simulator, whereas commercial implementations are available Cadence, Synopsys, and Siemens EDA. In addition, a limited subset of C++ and SystemC objects (version 1.4.7 for IEEE Std 1666-2011) can be used for High-Level Synthesis (HLS) [2].

SystemC supports system-level modeling at different abstraction levels using a wide range of models of computations, such as shared memory, message passing or discrete events. A detailed system-level modeling framework, often called a virtual platform (VP), can rely on a SystemC-based C++ Intellectual Property (IP) modeling library, a powerful simulation engine, a runtime and test environment, and refinement methodology. Two commercial tools for domain-independent SystemC-based design were developed in the early days: VCC from Cadence [5] and Synopsys Cocentric System Studio with SystemC Compiler [6]. Nowadays, new tools, such as Synopsys Platform Architect, Cadence Stratus HLS, and AMD's Vivado Design Suite (former Xilinx Vivado), enable early design space exploration and performance/power optimization of SoC architectures, which is not feasible in traditional RTL to GDS design flow, and High-Level Synthesis. In addition, major industries in EDA, semiconductor, design-reuse, and electronic systems use SystemC not only to perform architecture exploration, but also to develop and test high quality IP, engage in early development of embedded applications and system software, or deliver virtual platforms that facilitate hardware/software co-design.

Despite system design efforts, there is not yet an efficient SystemC-based

development environment capable of addressing challenges due to the increasing SoC design complexity. This environment should

- combine the necessary architecture/technology libraries, and synthesis tools to support user-defined specification of SystemC models, integration, validation, and rapid full system prototyping;
- enable IP reuse at the system level to provide significant savings in project costs, time scale, and design risk [7] [8]; and
- support efficient design space exploration and optimization.

In this chapter, we focus on general methodologies for system-level modeling, performance evaluation, and design space exploration, a key process for enhancing design quality.

More specifically, in Section 12.2, we consider system modeling, design flow, and design space exploration. We introduce abstraction levels and outline system-level design methodology, including

- top-down and bottom-up protocol refinement,
- hierarchical modeling, i.e., decomposing system functionality into subordinate modules (blocks and subblocks),
- orthogonalization of concerns (i.e., separating module specification from architecture, and behavior from communication), and
- communication protocol layering.

In Section 12.3, we outline SystemC-based system-level modeling objects, such as *module, clock, intra-module memory, intra-module synchronization,* and *inter-module communication channels,* and examine *back-annotation of different operational characteristics* of system-level models, including time-, power-, and security-based annotations for electronic design integration, providing a short tour of the newly proposed IEEE P3164 standard.

Then, in Section 12.4, we discuss examples of system-level modeling and related back-annotation issues. We consider the time-annotation of clock-accurate models by providing an example based on processor architectures. We also examine a more complex example of system-level modeling and time-annotation on a user-defined SystemC-based hierarchical finite-state machine (`HFsm`). This object provides two descriptions of model behavior; using *states, conditions, actions, and delays* as in traditional (flat) finite-state machines, or *states and events*. In addition, we examine security asset definitions related to a simple Watchdog Timer module.

We also examine inter-module communication channels in SystemC, and PLL design in SystemC-AMS, the SystemC extension for Analog Mixed Signal modeling.

In Section 12.5, we focus on system-level modeling methodologies for collecting performance statistics from modeling objects, including automatic extraction of statistical properties. Advanced high-level performance modeling environments may be based on advanced system-level monitoring activities. Thus, we examine the design of system-level tools, *generating, processing, presenting, and disseminating* system monitoring information.

In Section 12.6, we discuss open system-level modeling issues, such as asynchronous processing, parallel system-level simulation, and interoperability with other design tools. By resolving these issues, SystemC-based system-level modeling can achieve a higher degree of productivity.

12.2 Design Methodology

We examine fundamental concepts in system-level design, including abstraction levels, system-level modeling, and design exploration.

12.2.1 Abstraction Levels

A fundamental issue in system design is model creation. A model is a concrete representation of IP functionality. In contrast to component IP

models, a *virtual platform prototype* refers to system modeling. A virtual platform enables design integration, simulation, and validation of system functionality. It also supports reuse at various levels of abstraction, hardware/software partitioning, and design space exploration for different implementations, whether small- or large-scale.

A virtual platform prototype consists of:

- Models of hardware components, including peripheral IP block models (e.g., I/O, timers, Audio/Video codecs, or DMA), processor emulator via instruction set simulator (ISS) (e.g., ARM A-, M-, and R-series processors, such as A73, or open source RISC-V cores), and communication network with internal or external memory, including bus protocols, such as ARM AMBA AHB, APB, or AXI systems, and network-on-chip, such as ARM Neoverse CMN-700 coherent mesh interface.
- System software, including hardware-dependent software, models of RTOS, device drivers, and middleware.
- Environment simulation, including application software, benchmarks, and stochastic models.

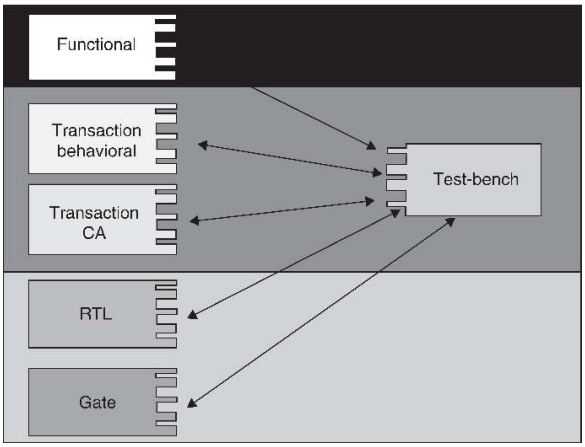


FIGURE 12.1 IP modeling in various abstraction levels.

Notice that a virtual prototype may hide, modify, or omit system properties. As shown in Figure 12.1, abstraction levels in system modeling span multiple levels of accuracy ranging from functional to transactional to clock- and bit-accurate to gate level. Each level introduces new model details. We now provide an intuitive description of abstraction levels, from the most abstract to the most specific.

- Functional models are appropriate for concept validation and partitioning between control and data. However, they have no notion of resource sharing or time, i.e., functionality is executed instantaneously, and the model may or may not be bit-accurate. This includes the definition of abstract data types, specification of hardware or software (for instance, RTOS) computation, communication, and synchronization, and algorithm integration to high-level simulation tools and languages, e.g., MATLAB, or UML.
- Transactional behavioral models (simply noted as transactional) are functional models mapped to a discrete-time domain. Transactions are atomic operations with their duration stochastically determined, i.e., several clock cycles in a synchronous model. Although detailed timing characteristics on buses cannot be modeled, transactional models are fit for modeling pipelining, RTOS scheduling, basic communication protocols, test-bench realization, and preliminary performance estimation. Transactional models are usually hundreds of times faster than lower-level models and are therefore appropriate for concept

validation, virtual platform prototyping, design space exploration, and early software development. SystemC currently integrates the TLM 2.0 communication standard for transaction-level modeling which supports blocking and non-blocking interface functionality and aims at model interoperability and reuse across different use cases. In addition, TLM 2.0 transactions can be annotated with more accurate system timing to obtain an approximately timed model that provides compared to a loosely timed functional model.

- Clock-accurate models (denoted CA) are behavioral hardware models that enforce accurate timing on all system transactions. Thus, synchronous protocols, wire delays, and device access times can be accurately modeled. Using discrete-event systems, this layer allows for simple, generic, and efficient clock-accurate performance modeling of abstract processor core wrappers (called bus functional models), bus protocols, signal interfaces, peripheral IP blocks, instruction set simulators and test benches. Time delays are usually back-annotated from register transfer-level (RTL) models since clock-accurate models are not always synthesizable.
- Clock-accurate and bit-accurate models (CABA) are clock-accurate models that are also bit-accurate internally and at the interface level. CABA models offer an efficient, attractive path to implementation via automatic hardware/RTOS synthesis (High-Level Synthesis, or HLS), freeing the designer from timing and performance decisions while allowing for high reusability of the design and its test bench.
- RTL models correspond to the abstraction level from which synthesis tools can generate gate-level descriptions (or netlists). Register-transfer-level systems are usually visualized as having two components: data and control. The data part consists of registers, operators, and data paths, while the control part provides the time sequence of signals that stimulate activities in the data part. Data types are bit-accurate, interfaces are pin-accurate, and register transfer is clock-accurate.
- Gate models are described in terms of primitives, such as Boolean logic with timing data and layout configuration. For simulation reasons, gate models may be internally mapped to a continuous-time domain, including currents, voltages, noise, clock rise, and fall times. Storage and operators are broken down into logic implementing the digital functions corresponding to these operators while timing for individual signal paths can be obtained.

Thus, according to the above, as an example, an embedded physical SRAM memory model may be defined as:

- a high-level functional model described in a programming language, such as C or C++;
- a clock-accurate (and possibly bit-accurate) model, allowing validation of its integration with other components;
- implementation-independent RTL logic in HDLs, such as VHDL;
- a vendor gate library described using NAND, flip-flop schematics; or,
- a detailed and fully characterized mask layout at the physical level, depicting rectangles representing gate and transistor on chip layers, and a geometrical arrangement of I/O and power locations.

12.2.2 System Modeling Methodology

System modeling methodology involves a combination of *stepwise protocol refinement, hierarchical modeling, orthogonalization of concerns, and communication layering* techniques.

Stepwise protocol refinement is achieved through a combination of top-down and bottom-up approaches.

- In *bottom-up refinement*, IP-reuse-oriented integration with optimal evaluation, composition, and deployment of prefabricated or predesigned IP block and system components drive the process.
- In *top-down refinement,* emphasis is placed on specifying capturing system requirements with unambiguous semantics, performing optimal partitioning of system behavior into simpler component behaviors, and gradually refining the abstraction level down to a concrete low-level architecture. This process requires adding details and constraints in a narrower context while preserving desired properties. Top-down refinement allows the designer to explore modeling at various levels of abstraction, thus trading model accuracy with simulation speed.

This continuous re-arrangement of existing IP in ever-new composites is a key process to new product ideas. It also allows for extensive and systematic reuse of design knowledge and application of formal verification techniques. The concrete architecture obtained through formal refinement must satisfy the following properties:

- relative correctness, i.e., it must logically imply the abstract architecture by conforming to system specifications, and
- faithfulness, i.e., no new rules or constraints can be derived during the refinement process.

While formal refinement is hard to achieve since there are no automated translation techniques, relative refinement is based on *patterns* consisting of a pair of architectural schemas that are correct for a given mapping. By applying refinement patterns, e.g., a state transformation in a control flow graph [9], we can systematically and incrementally transform an abstract architecture to an equivalent lower-level form. Notice that formal architectural transformation is related to providing an augmented calculus with annotations for properties such as correctness, reliability, and performance [10].

Hierarchy is a fundamental ingredient for modeling conceptual and physical processes in systems of organized complexity using either a top-down analytical approach, i.e., the divide and conquer paradigm, or a bottom-up synthesis approach, i.e., the design-reuse paradigm. Hierarchical modeling based on simpler, subordinate models controlled by high-level system models enables a better encapsulation of the design's unique properties, resulting in efficient system design with simple, modular, and efficient interfaces.

For example, hierarchical modeling of control elements, memory, network components, and test benches enables the system model to become transparent to changes in component behavior and communication. A systematic framework supporting hierarchical modeling can be based on the SystemC module object, as well as the inheritance and composition features of the C++ language [11] [12]. Thus, SystemC modules consist of SystemC submodules in nested structures. Moreover, SystemC facilitates the description of hierarchical systems by supporting the module class hierarchy efficiently.

Stepwise refinement of a high-level behavioral model of an embedded system into actual implementation can be based on the concept of *orthogonalization of concerns* [13]:

- Separation of functional specification from final architecture, i.e., what the basic system functions are for system-level modeling vs. how the system organizes hardware and software resources to implement these functions. This concept facilitates system partitioning and hardware/software co-design into GNU/Linux or RTOS target code, or synthesizable RTL by focusing on progressively lower levels of abstraction.
- Separation of communication from computation (called behavior) enables plug-and-play system design using blocks that encapsulate

the desired data communication model. This concept reduces ambiguity among designers and enables efficient exploration.

Layering simplifies individual component design by defining functional entities at various abstraction levels and implementing protocols to perform each entity's task and their interactions. Advanced inter-module communication refinement may be based on establishing distinct network layers. This simplifies the design and maintenance of communication systems [14] [15] [16]. In most cases, we adopt two layers:

- The low-level communication layer provides a generic message API, abstracting away the fact that there may be a point-to-point channel, a bus, or an arbitrarily complex network-on-chip.
- The driver layer builds the necessary communication channel adaptation for describing high-level protocol functionality, including error handling, compatible protocol syntax, i.e., packet structure, and semantics, such as temporal ordering.

12.2.3 Design Space Exploration

To evaluate the vast number of complex algorithmic, architectural, and technological interactions, the architect must be equipped with a highly automated, efficient design space exploration methodology. This methodology can construct an initial design solution from system requirements, mapping modules to system resources. This solution can be subsequently refined through an iterative strategy that optimizes performance, power, reliability, and resource contention criteria using domain- or application-specific *stochastic models and analysis tools,* or *system benchmarks*, e.g., networking or multimedia. The proposed solution must select system parameters, including configuration for sophisticated multithreading, prefetching, and memory hierarchy components.

After generating an optimized mapping of behavior and communication onto an initial system architecture, the designer may manually translate hardware components to the register transfer level of abstraction. Alternatively, he/she may load system-level schematics (code and configuration parameters) onto behavioral synthesis tools, such as Synopsys Platform Architect and AMD Vivado Design Suite. These tools integrate a custom design with a pre-loaded library of configurable, high-level (soft) IPs, e.g., processor cores, memories, peripherals, and bus or network-on-chip interconnects. The target is to automate the synthesis of IPs using an efficient strategy. Then, the design flow can proceed normally with place and route, simulation, and optimization using tools, such as Synopsys IC II Compiler, for time/power analysis, and physical design signoff.

12.3 Annotation of System-Level Models

SystemC allows either clock-accurate bit-accurate modeling or clock-approximate transaction-level modeling. In this section, we consider back-annotation of system-level models, focusing on hardware modeling objects.

12.3.1 System-Level Modeling Objects

As a prelude to back-annotation, we examine a high-level IP modeling environment using the SystemC-based C++ modeling library and associated simulation kernel. A proprietary runtime and test environment may be provided externally. This high-level modeling environment supports user-defined C++ building blocks (macro functions) that simplify system-level design, enable controlled concurrency (supported by synthesis), and provide the means for efficient design space exploration and hardware-software partitioning.

High-level SystemC-based modeling involves executable specifications of hardware objects as well as system and application software components. For models of complex real-time systems, including early software development and verification, software components may include calls to safety critical, RTOS models [17] [18]. These library calls, when properly annotated, allow measuring the impact of different scheduling policies, communication & synchronization libraries, and external drivers. An RTOS model abstracts a real-time operating system by providing generic system calls for system management (RTOS kernel scheduling and initialization), application-level task management and synchronization (fork, join, create, sleep/wakeup, mutex/spinlock, semaphores), process communication, and asynchronous event handling (wait, signal notify). Future SystemC versions proposed by the SystemC Language Working Group are expected to address full system prototypes running RTOS, and user-defined scheduling constructs on top of the core SystemC scheduler. During software synthesis, an RTOS model may be replaced with an open-source or commercial RTOS.

Focusing on hardware, we describe system-level common modeling objects (components) in SystemC models. Most objects fall into two main categories: *active* and *passive modeling objects*. While active objects include at least one thread of execution, initiating actions that control system activity during simulation, passive objects do not have this capability and only perform standard actions required by active objects. Notice that communication objects can be active or passive, depending on the abstraction level. Each hardware block is instantiated within a SystemC class called `sc_module`. This class is a container of a user-selected collection of SystemC clock domains, active computation, control flow, communication models, and passive memory objects including:

- *SystemC and user-defined data types* providing low-level access, such as `Bits` and `Bytes`, allowing for bit-accurate, platform-independent modeling.
- *User-defined passive memory objects* such as `Register`, `FIFO`, `LIFO`, `circular FIFO`, `Memory`, `Cache`, and user-defined collections of externally addressable hierarchical memory objects.
- *SystemC and user-defined intra-module communication and synchronization objects* based on message passing, such as `Monitor`, or concurrent shared memory such as `Mutex`, `Semaphore`, `Conditional Variables`, `Event Flag`, and `Mailbox`. This includes `Timer` and `Watchdog Timer` for generating periodic and nonperiodic time-out events.
- *Active computation and control flow objects*, such as SystemC processes: C-like functions that execute without delay (`SC_METHOD`), asynchronous threads (`SC_THREAD`), clocked threads (`SC_CTHREAD`), and complex, user-defined objects, such as hierarchical finite machines (`HFsm`). `HFsm` captures abstract processor models using instruction set simulators, or custom techniques, e.g., cross-compiling the (bare metal) application code, mapping macro- to micro-operations, and annotating timing data based on a technical reference manual (e.g., timing delays for ARM Cortex-A9 [19]) or reverse profiling techniques.
- *Inter-module communication object*, e.g., based on a message passing, shared memory, or hybrid model. Note that intra-module interaction, including computation, communication, and synchronization, is performed either by local processes or control flow objects defined in remote modules.
- *User-defined inter-module communication channels (point-to-point or multi-point)* and corresponding interfaces, e.g., AMBA bus (AHB, APB, AXI). These objects are simply built on top of SystemC. For example, OCCN provides an open-source, object-oriented library of objects built

on top of SystemC `sc_channel` with the necessary semantics for modeling on-chip communication infrastructure [15], while Heterogeneous System on Chip (HSoC) provides a similar library for SoC modeling [20]. Notice that SystemC components can be integrated with other SystemC-based libraries and tools already available in the market, such as a variety of open processor architecture models in Synopsys/Imperas OVP suite [21].

The latest SystemC version 3.0.1 ensures compliance with IEEE Std 1666-2023. The update includes a variety of enhancements related to coding style and performance, asynchronous reset events, C++ data types (from the fundamental C++ layer), VCD waverform tracing, bug fixes, and expanded platform support.

12.3.2 Time Annotation

When high-level views and increased simulation speed are desirable, systems may be modeled at the transactional level using an appropriate abstract data type, e.g., processor core, video line, or network communication protocol. With transactional modeling, the designer can focus on IP functionality rather than on details of the physical interface, including data flow, FIFO sizes, or time constraints. For efficient design space exploration, transactional models (including application benchmarks) must be annotated with several architectural parameters.

- Realizability parameters capture system design issues that control system concurrency, such as processor, memory, peripherals, network interface, router, and their interactions, e.g., instruction execution delays, arbitration, packetization, and packet transfer. Also, other parameters may be considered: RTOS model parameters for context switch or operation delays, and VLSI layout parameters for time-area tradeoffs, clock speed, bisection bandwidth, and power consumption models.
- Serviceability parameters refer to reliability, availability, performance, and fault-recovery models for transient, intermittent, and permanent hardware/software errors. When repairs are feasible, fault recovery is usually based on detection (through checkpoints and diagnostics), isolation, rollback, and reconfiguration. While reliability, availability, and fault-recovery processes are based on two-state component characterization (faulty or good), performability metrics evaluate failsafe system performance degradation in the presence of faults, e.g., increased latency due to packet congestion when there is limited loss of network connectivity.

Timing information is only one of many types of annotations. To provide accurate system performance measurements, e.g., power consumption, throughput rates, packet loss, latency statistics, or QoS requirements, all transactional models involving computation, communication, and synchronization components must be back-annotated with an abstract notion of time. Thus, timing characteristics, obtained either through hardware synthesis of clock-accurate hardware models or evaluated by profiling software code on an RTOS-based embedded processor, can be back-annotated to the initial TLM model. The clock-accurate system architecture model can be used for system-level performance evaluation, and design space exploration. It can also serve as an executable specification for system software implementation.

As experienced in many commercial and open environments, such as ARM Fast Models platform, QEMU, and the gem5 simulator, hardware delays are not always clock-accurate, but rather clock-approximate. For example, for computational components mapped to a particular RTOS, or large communication transactions mapped to a particular shared bus, it is difficult to estimate accurate delays that depend on precise system

configuration and load. Moreover, for deep submicron technology, wire delays that dominate protocol timings cannot be determined until layout time. Thus, to avoid system hazards, and ensure correct behavior independent of computation, communication, and synchronization delays, one must include all necessary synchronization points and interface logic in the transactional models. Analysis using parameter sweeps helps estimate the sensitivity of system-level design due to perturbations in the system architecture, and thus examine the possibility of adding new features in derivative products [22].

For most SystemC-based hardware objects, time annotation is performed statically, during instantiation time. Note that time delays also occur dynamically due to interactions of the underlying protocol semantics. As an example, we now consider memory and communication channel objects; `HFsm` and concurrent `HFsm` (`HCFsm`) are discussed in the next section.

- Time is annotated during the instantiation of memory objects by providing the number of clock cycles (or absolute time) for respective read/write memory operations or bursts. The timing semantics of the memory access protocol implemented within an active control flow object may contribute to additional dynamic delays. For example, a *congestion delay* may be related to allowing a single-memory operation to proceed within a particular time interval, and a *large delay* may be due to transferring only a small amount of information per time unit due to network contention. CA modeling of refresh operations after writing to memory requires a complex, annotated finite state machine with tens of parameters [23].
- Time must be annotated during the instantiation of the processes attached to the communication channels by implementing timing constraints in blocking or nonblocking calls and providing SystemC statements `wait(n)` possibly within `do {…} while (!valid);` hence, in its simplest form, the SystemC process executes `n=compute_function(); wait(n)`, where `n` is the time to perform the same behavior on the actual hardware. Note that we can model timing in an instruction-accurate, "faithful" way (as described before), or using multiple instructions followed by a special `sync` statement that produces the `wait` statement. Blocking calls within inter-module communication objects instantiated within the SystemC channel object or adherence to specific protocol semantics may imply additional dynamic delays, e.g., compatibility in terms of the size of communicated tokens, or packet loss based on network quality criteria; these parameters may be annotated during inter-module communication channel instantiation; this annotation may also invoke related intra-module communication/synchronization objects.

12.3.2 Power Intent

As the IC packing density increases, it is fundamental to reduce power by examining power management in early stages of the design flow. Therefore, system-level hardware models must be annotated with power consumption information [24] [25]. The model may later be refined to include detailed power information, such as voltage and supply nets.

More specifically, power state models are used in CPU, memory, and hardware accelerators. They evaluate static and dynamic power dissipation for each possible transition of a hardware component, such as between processor instruction pairs, or between read, write, and refresh memory operations. Thus, the system-level model is power-annotated by inserting custom activity transitions, such as `set_state(activity, type, wakeup_time)`, where `type` can be idle, wait, running, etc. Moreover, accurate estimation of memory power consumption requires trace-checkers and traffic distribution analyzers, since power is affected by many application

and architectural parameters, such as access patterns, or controller logic.

Alternatively, power in communication systems, from internal buses and networks-on-chips [26] to external wireless components and network processors [27], is modeled using Joule-per-bit traffic models. Realistic consumption profiles reveal that power consumption depends on the number of transactions (related to clock frequency), the number of bits per transaction, and the cost of transmitting a single bit (energy cost per bit). Hence, the number of transactions (`T`) can be obtained by instrumenting each SystemC function, e.g., `(n, T)=compute_function(); wait(n);`

Power management treats static power leakage using power gating and DVS techniques, and dynamic power management using clock gating and dynamic frequency scaling.

While clock-tree modeling, clock gating, and DFS are currently performed directly on the system model, to achieve reusability of system-level IP concerning power intent, one can rely on Accellera's Unified Power Format (now IEEE Std 1801) [28] or Si2's Common Power Format. In fact, UPF 3.1 provides specifications for electronic system-level power intent. UPF helps express multiple power domains, voltage levels, isolation/retention cells, and level shifters. It also helps address power gating and Dynamic Voltage Scaling (DVS) since UPF 3.0+ does not include clock-tree modeling. In addition, Si2's Unified Power Model (now IEEE Std 2416) helps in formalizing the methodology for power modeling [29], i.e., feeding data to the UPF system-level power model for analysis, management, and optimization. UPM supports multiple power models (equations, tables, or actual characterization numbers) appropriate for different abstraction levels, including system-level architecture design. Similar to security annotations, UPM semantics use XML elements and attributes.

Commercial tools, such as Synopsys Platform Architect and Intel Cofluent Studio, and open-source power estimation tools instrument SystemC IP models with power state and traffic models, using class overloading and augmented channels [30] [31] [32].

In addition, some frameworks implement C++ classes to represent UPF in SystemC-based design. SCPower [33] implements synthesizable power domains and isolation to enable power-aware system-level simulation. PMS [34] combines the placement of synthesizable power intent specifications and synthesizable power management directives directly in the SystemC model to control the active power state for each power domain in a given operation mode. On the other hand, LP-HLS [35] uses compiler pragmas in the SystemC model to support synthesizable power intent and power-gating directives in the less popular CPF format. It can also perform power management by switching between on/off power states in the power domains.

In this context, the PwClkARCH tool by CNRS and NXP focuses on UPF power-intent semantics, enabling continuous power monitoring and management on NXP hardware [36] [37] [38]. The tool supports power estimation for approximately timed TLM models, including power state and traffic models, interacting with DRAMpower for modeling memory power consumption [39]. It also helps define clock intent, power domains, internal or external clocks, DPLLs, clock managers, and operating point tables with legal combinations of clock/power states. The tool has been used to model power in complex systems specified at system-level with up to 95% accuracy, such as power-aware switch matrix displays with two external DRAMs and multiple clock and power domains, Intel's clock-gating, and power-gating in custom hardware accelerators. The design flow allows modeling power of AXI-based initiator/target systems with SystemC callback threads activated upon transaction receipt.

Leakage current increases exponentially with temperature [40]. Hence, power-thermal co-simulation of a timed SystemC TLM model is used by instrumenting the SystemC model with power management and a thermal

equation solver (ATMI, HotSpot, or industrial one) [41]. The solver input is a preliminary floorplan and the power consumption of each component [42] [43] [44]. Both power state models and traffic models are supported.

12.3.4 Security Annotation

Modern SoCs often have significant security vulnerabilities that are extremely exposed during integration. They may permit third-party hardware or software IPs to be used as vehicles for malicious attacks that interfere with the IP's multiple subsystems, interfaces, zones, or parameters. To avoid complex redesign costs, it is crucial to address security issues or flaws in early system-level design [45]. This requires reconsidering integration patterns and IP reuse practices for hardware IPs by focusing on security assets, weaknesses, and mitigation policies for prevention, protection, and resilience, thus elevating security by design as a first-order design constraint.

Thus, quite different from time annotation, which leverages clock-accurate time characteristics through hardware synthesis or software profiling to back-annotate the initial TLM model for high-speed simulation and architecture exploration, security annotation aims to identify security assets in `sc_module` (e.g., a custom SystemC function block, or an ASIC IP from a third party). More specifically, security annotation aims to collect all security-concern-related objectives by identifying a) critical assets, b) elements with their attributes like ports, parameters, and conditions to accomplish a security objective, c) weaknesses, such as confidentiality, integrity, availability, isolation, and leakage of side or covert channels, and d) a link to a Common Weakness Enumeration (CWE) database that provides potential mitigation policies [46] [47].

Security annotations of sensitive IP must be examined during design exploration (since security is generally considered a QoS parameter), system integration, and software/firmware development by creating a threat model with the specified mitigations to support the security objective. The IEEE Std P3164 aims to advance the SA-EDI (Security Annotation for Electronic Design Integration) standard [48]. Using this provisional standard, IP designers can identify security concerns, so that EDA tools can help mitigate weaknesses, or disclose related information to the higher-level integrator for possible actions.

As shown in Figure 12.2, the IEEE P3164 standard develops a security annotation methodology for designers and EDA tools in a generic ASIC subsystem integration or rapid prototyping flow. The annotation includes data objects representing Assets, Database entries, Elements, Weaknesses (or Attack Points), and security objectives that define an intent to mitigate the identified threats.

The Asset data objects identify sensitive (or critical) material within the specific `sc_module` subsystem. Assets are related to Weakness objects in a CWE Database and are input into EDA tools that create corresponding Element data objects.

The Element data objects contain ports and parameters that influence and/or observe the behavior of the asset and include potential security weaknesses associated with the asset. These objects help construct the Weakness data objects (or so-called attack point security objectives, or APSO).

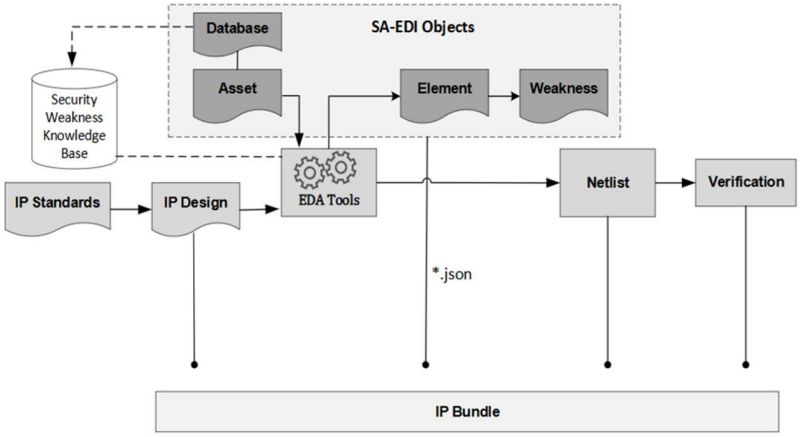


FIGURE 12.2 Security annotation: design flow.

APSO objects associate a potential security concern to an asset by defining the ports and/or parameters from Element objects. APSO objects depend on the architecture and implementation of the module (subsystem) as a standalone component. Additionally, the owner could create APSO objects based on potential use cases of the module in an integrated system, such as a SoC. APSO objects are used by the system integrator to specify the threat model, at the system or IP level.

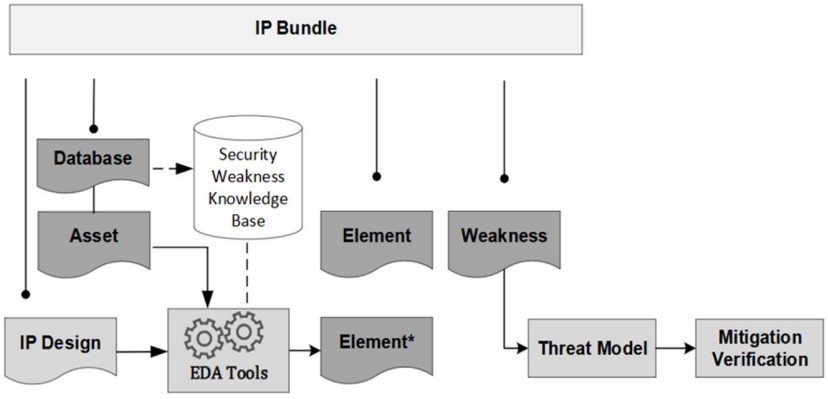


FIGURE 12.3 Security annotation: data objects.

All objects in Figure 12.3 are specified in JSON format and included in the IP Bundle as a complete third-party IP. Moreover, the Element objects may be created manually, without an EDA tool. The database is a "Security Weakness Knowledge Base" based on industry experience and/or security research. The standard allows the use of different databases from multiple sources, e.g., MITRE [49] or NDS [50].

The security annotation format collects the SA-EDI standard's data objects into a single JSON file [48]. Although this is optional, if provided in the IP Bundle, the schema defines the asset, element, database, and APSO (weakness) objects. Since the general format for each object is undergoing an update, please refer to the IEEE Design Automation standards on Security for the upcoming version.

12.4 SystemC Models and Back-Annotation

Modeling and design space exploration of complex embedded systems that combine CPUs, memories, interconnects, and I/O peripherals is a tedious task. Thus, system designers must be equipped with an architecture simulation tool capable of converting quickly and without much effort high-

level concepts to a specific model. In this context, different interoperable system-level design tools provide simulation efficiency vs. accuracy tradeoffs that enable exploring a subset of the design space.
Next, we consider five short case studies related to system-level modeling and back-annotation: an abstract processor model, a hierarchical final-state machine, an asynchronous channel in SystemC, a phase-lock loop in SystemC-AMS, and security annotation of a Watchdog Timer module.

12.4.1 Abstract Processor Models

An abstract processor model for an existing multicore architecture can use co-simulation, e.g., ARM Cortex-A9 co-simulation in the popular Zynq 7000 reconfigurable SoC. Using this approach, we can simulate the performance of a bare metal application, i.e., without using an Operating System, in a clock-approximate manner. Notice that full system simulation (e.g., under Linux) is not strongly supported by SystemC. Although there is multiprocessor support in some high-level design space exploration frameworks supporting tens of Instruction Set Architectures, such as the Synopsys/Imperas OVP framework [21], interoperability with SystemC is at an early stage and requires extra effort. The same holds for QEMU, an open-source full-system virtualization and emulation platform targeting x86, ARM, RISC-V, PowerPC, Xtensa, and Tricore systems. However, these concepts bring QEMU and SystemC closer to coupling (or eventually merging).

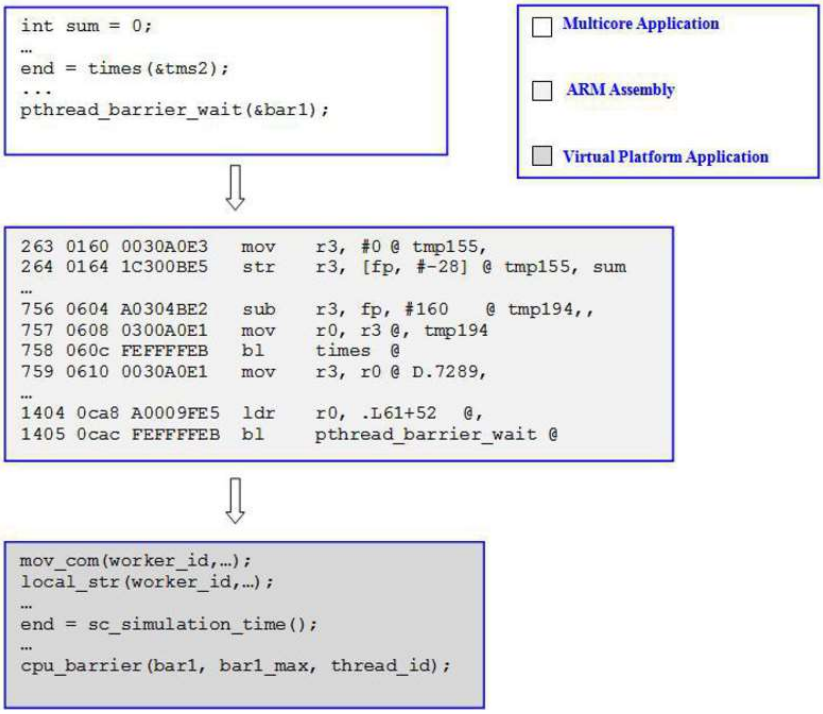


FIGURE 12.4 Program conversion steps for ARM co-simulation.

For the open source Gem5 simulator [51] [52], the proposed master-slave transactor mechanisms proposed in [53] translate gem5 memory accesses and timing to SystemC TLM 2.0 non-blocking transactions and vice-versa. Moreover, the effectiveness of asynchronous TLM 2.0 channels for real-time co-simulation of SystemC models with actual platforms, i.e., bare-metal or full-system prototypes is shown in [54] and [55].

To start with, we can model each ARM Cortex-A9 processor instruction in the execution pipeline (e.g., `add`, `subtract`, `multiply`, `compare and`

`branch`, `move`, `shift`, and `local load/store`) in SystemC at the transaction level, and then statically back-annotate available clock-approximate timing information from the assembly instruction cycle delays. Accurate timings are in the technical architecture reference manual for Cortex-A9 [19]. Hence, each processor instruction can be modeled as a SystemC function, which contains a `wait(x);` statement, where `x` is the number of processor cycles, back-annotated from the previously mentioned reference manual.

The application test bench should be written in C/C++ and compiled using an ARM compiler (or cross-compiler) with flags suitable for generating assembly code. As a second step, this assembly code is replaced by the corresponding processor instructions modeled in an abstract SystemC-based processor model. In other words, each assembly instruction is mapped to a SystemC function, which is back-annotated with the corresponding delay. Notice that to follow the exact trace of the original program (when it branches), the equivalent C++ instruction is also executed by the simulator.

Figure 12.4 shows the conversion from application code to assembly, and from assembly to SystemC function calls.

Modeling instruction and data pipeline, branch prediction, and caching effects require knowledge of the network-on-chip architecture, and the memory hierarchy, including organization, and cache line size [56]. Model inaccuracies for instruction cache can arise due to indirect addressing and the associated branch prediction model of the processor architecture. Detailed modeling slows down simulation but provides an accurate execution time estimation for a given ISS microarchitecture. RISC-V ISA extensions are modeled by extending the decode and execute functions to take into account transaction information, including flow (order from an initiator to a target, reference address, time annotation), data attributes (command, address, data value/length, response status), and transaction type (loosely timed or approximately timed) [57] [58].

12.4.2 Hierarchical Finite-State Machines using SystemC

Finite-state machines are critical for modeling dynamic aspects of hardware systems. Most hardware systems are highly state-dependent. Their actions depend not only on their current inputs but also on what has previously happened. Thus, states represent a well-defined system status over a time interval, e.g., a clock cycle. In cycle-accurate simulation, system events occur within a cycle. In this way, we can capture the correct semantics, and back-annotate hardware objects in a detailed and accurate way.

Hierarchical finite-state models based on Harel state charts and variants define complex, control-dominated systems as groups of smaller subsystems, permitting the model developer to consider each subsystem as a separate module [59] [60]. Hierarchy is commonly disguised into two forms:

- A structural hierarchy enables the realization of new components based on existing ones. For example, a system is a set of interconnected components specified at various abstraction levels.
- Behavioral hierarchy enables system modeling using sequential (e.g., procedure or recursion) or concurrent computation (e.g., parallel or pipelined decomposition).

Like asynchronous and clocked threads (`SC_THREAD` and `SC_CTHREAD` constructs) defined in SystemC, an `HFsm` modeled in C++/SystemC is a powerful mechanism for capturing a description of dynamic behavior within a SystemC model. `HFsm` may be used to capture formally the dynamic behavior of a hardware module, e.g., a circuit representing an embedded system controller or a network protocol. For control tasks, implementations of asynchronous or clocked threads in SystemC and `HFsm` are equivalent.

Although clocked threads are usually simpler and more efficient from the simulation time point of view since the sensitivity list is just the clock edge, an `HFsm` implementation is more general, providing clarity and efficiency when specifying complex control tasks. It also allows for `HFsm` back-annotation in a more detailed way, thus making CA modeling easier to achieve. Since outer `HFsm` states are typically behavioral generalizations of inner ones, `HFsm` facilitates the implementation of policy mechanisms, such as signaling exceptions and pre-emptive events. It also avoids pitfalls in implementations of traditional (flat) finite-state machines, such as:

- Exponential explosion in the number of states or transitions when composing substates since the model uses less redundant timing information.
- Difficulty in modeling complex hierarchical control flows, with transitions spanning different levels of the behavioral hierarchy.
- Difficulty managing transitions in hierarchical groups and global control signals specified for a set of states, e.g., reset and halt.
- Limited support for concurrency.

12.4.2.1 Flat HFsm Description: States, Conditions, Actions & Delays

A flat `HFsm` is specified using simple, common modeling constructs: states, conditions, delays, and actions. States identify `HFsm` status, conditions drive actions, and delay times postpone actions. Composite transitions are quadruples formed by a condition, an action, a delay, and the next state. A flat `HFsm` is represented graphically in Figure 12.5.

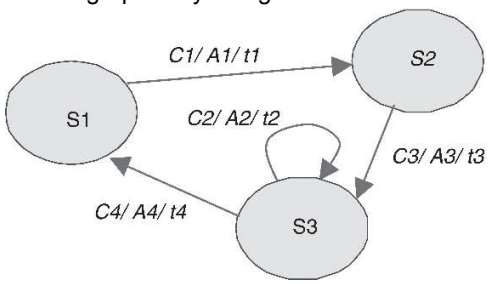


FIGURE 12.5 Graphical illustration of a flat, clocked finite-state machine.

An equivalent representation is as follows.

- State `S1={{C1, A1, t1, S2}};`
- State `S2={{C3, A3, t3, S3}};`
- State `S3={{C4, A4, t4, S1}, {C3, A3, t5, S3}};`

Let us consider the state `S3`. In this state, we have two transitions. In the first one, `S1` is the next state, `C4` is the condition, `A4` is the action, and `t4` is a constant delay. The semantics of this transition are as follows. If the `HFsm` is in state `S3` and condition `C4` is true, then `HFsm` performs action *A4* and transfers control to state `S1` after a delay of *t4* clock cycles. Delay annotation during state changes is common in `HFsm` specifications, e.g., when examining the dynamic behavior of objects, such as a processor or memory controller. Moreover, the `HFsm` designer could redefine this order, e.g., condition-delay-action instead of condition-action-delay.

Furthermore, if `HFsm` conditions were *not* allowed to make blocking function calls, the delay associated with performing the corresponding action would be *deterministic* and could be estimated and back-annotated directly by the model designer. *Variable delays* due to concurrent accesses to shared resources, cause non-deterministic delays. These delays are much harder to model accurately. They can be accounted for in the action routine by placing appropriate blocking function calls to time-annotated intra-module communication and synchronization objects, such as user-defined channels.

These channels must accurately emulate system synchronization conditions.

12.4.2.2 HFsm Description using States and Events

For complex examples, hierarchical states may have “parents” or “children” at distinct levels. This modeling approach is frequent with machines using different operating modes, i.e., machine features that define alternative configurations. Then, `HFsm` can be expressed conveniently using discrete-event systems, i.e., a finite set of states, a finite set of event categories, startup and regular state transitions mapping (state, event) pairs to other states, and actions associated with transitions. Events are triggered by changes in conditions, which (like actions) can be time annotated. This description is natural, abstract, powerful, and efficient.

- Initial startup transitions originate during initial transitions from a top superstate. These transitions invoke entry and startup actions (e.g., setting state parameters) for the state associated with the transition and intermediate superstates if a transition crosses two or more levels of hierarchy.
- Regular transitions occur at the current state during `HFsm` processing. Transitions are triggered by events selected either randomly or through external stimuli from a user-defined enumerated event list. These transitions cause entry (and exit) actions upon entering (respectively, exiting) an `HFsm` state. Depending on the position in the state hierarchy of the new vs. the current state, entry or exit actions are executed from the least to the most deeply nested state, and vice versa upon exit. Entry and exit actions always precede processing actions associated with the newly entered hierarchical state.

Hierarchical finite state machines are common when describing operation control in digital systems when a *top menu* superstate is specialized to *different operating modes* (e.g., time and date keeping for a digital watch) and one or more *normal operating modes* (e.g., for text or video content in an on-screen display of a digital TV). These modes of operation can be considered independent superstates.

Within hierarchical system specification, we present a SystemC-related implementation of `HFsm` that clarifies the previous definition. To annotate time delays, SystemC `wait/wait_until` can be associated with each processed event; power annotation is also possible. Moreover, blocking calls to intra-module communication and synchronization objects may account for additional dynamic delays. These delays are annotated to startup transitions (`START_EVT`) from a superstate and subsequent state entry (`ENTRY_EVT`) or state exit (`EXIT_EVT`) transitions. Note that state entry or exit transitions may occur within a given hierarchical level or between different levels; hence, to discover which entry or exit actions to execute, it is necessary to compute dynamically, i.e., during state transitions, the least common ancestor (LCA) from the least deeply nested (source or destination) to the most deeply nested (destination or source).

12.4.3 Asynchronous Inter-Module Communication in SystemC

Unlike synchronous communication channels, where individual modules share a clock, asynchronous blocking communication channels use local handshake protocols to exchange data among nodes connected to memory-less wires. Globally asynchronous locally synchronous (GALS) techniques relax the synchrony condition by employing multiple non-synchronized clocks, achieving improved power consumption and reduced electromagnetic interference [61]. GALS systems may behave deterministically if system output cannot differ for the same input sequence. Deterministic GALS

systems must provide a unique output independent of frequency/phase variations, interconnect delays, and clock skew.

For example, in asynchronous and GALS inter-module communication, we consider two SystemC modules exchanging information over an asynchronous pipeline. Information is transmitted only if the sender is ready to send and the receiver ready to receive. Many kinds of SystemC channels represent this type of handshake. For clock-accurate and bit-accurate communication, the SystemC signal data channel provides the necessary event synchronization functionality, e.g., `wait(signal.event)`. However, as indicated in the following two subsections, particular care must be taken when implementing asynchronous or GALS communication using signals in SystemC, not only from an annotation perspective but also from a correctness viewpoint. In the closing section, we provide a solution to both problems based on the four-phase asynchronous handshake protocol which allows for back-annotation. Time annotation of an industrial network-on-chip model (STMicro's STNoC), without the formation of bubbles between packet flits during virtual cut-through routing, is examined in [47].

12.4.3.1 Asynchronous Inter-Module Communication Using SystemC Signals

We first consider typical communication between two modules connected via a SystemC signal of type `data_type`, i.e., the `data_in` port of one is connected to the `data_out` port of the other, and vice-versa. Without loss of generality, we assume that both modules have similar functionality and focus on their communication interface. The following code snippet helps make this case familiar to SystemC developers.

```
class device: public sc_module {
       public:
       device(sc_module_name nm):sc_module(nm) {
            SC_THREAD(in_action);
                  sensitive << data_in;

       }
       sc_in<data_type> data_in;
       sc_out<data_type> data_out;
       void in_action();
};
void device::in_action(){
       while(1){
            wait(); //waits an event in data_in port
            data_type module_data = data_in.read();
            // process received data
            ...
            // Write data to the other module
            data_out.write();

       }
};
int sc_main (int argc, char* argv[]) {
...
       sc_signal<data_type> signal_1;
       sc_signal<data_type> signal_2;
       device module1("Module1");
       module1.data_in (signal_1);
       module1.data_out (signal_2);
       device module2("Module2");
       module2.data_in (signal_2);
       module2.data_out (signal_1);
...
```

```
        sc_start();
}
```

In the above implementation, we present a two-way inter-module asynchronous communication, i.e., without clock synchronization. More specifically, when the sending module has data ready in its `data_out` port, the `data_in` port of the second module receives the data through the appropriate `signal_x` signal, where `x=1,2`.

This implementation is easy to use and has no clock dependency, so it supports inter-module communication in both synchronous and asynchronous systems, i.e., systems that use the same or different clocks. However, it is not always correct since there is a limitation in event generation related to SystemC signals. More specifically, a signal generates an event if and only if the signal changes its value. Thus, if a sender repeatedly executes the write method on its output port but writes the same value, only one write is triggered in the event list of the corresponding signal listening port, i.e., the second write event is lost.

12.4.3.2 Clock-Sensitive Inter-Module Communication

Next, we consider two modules, each with a clock-sensitive thread. The modules use identical or different clocks and interact with each other, like before, through signals. As in the previous section, we focus on communication interface definitions rather than the module functionality. The modules are again connected via a SystemC signal of type data_type, as shown below. The following code helps make this example more concrete.

```
class device: public sc_module {
        public:
        device(sc_module_name nm):sc_module(nm) {
              SC_THREAD(in_action);
                    sensitive << pkt_in << CLK;
        }
        sc_in_clk CLK;
        sc_in<data_type> data_in;
        sc_out<data_type> data_out;
        void in_action();
        };
void device::in_action(){
        while(1){
              //wait for event in pkt_in port
              wait(data_in.value_changed_event());
              data_type module_data = data_in.read();
              // process received data
              ...
              wait(x); //module can wait for x cycles
              // Write data to the other module
              data_out.write();
        }
};
int sc_main (int argc, char* argv[]) {
...
sc_clock clock1("CLOCK1", ZZ, SC_NS, X.X, Y.Y, SC_NS);
sc_clock clock2("CLOCK1", ZZ, SC_NS, X.X, Y.Y, SC_NS);
...
        sc_signal<data_type> signal_1;
        sc_signal<data_type> signal_2;
        device module1("Module1");
        module1.data_out (signal_1);
        module1.data_in (signal_2);
        module1.CLK (clock1);
```

```
        device module2("Module2");
        module2.data_out (signal_2);
        module2.data_in (signal_1);
        module2.CLK (clock2);
...
        sc_start();
}
```

This implementation not only suffers from the same SystemC limitations as the previous example requiring uniqueness of the signal values, but since the listening thread `in_action` is sensitive to both the data signal and the clock, it is possible that while waiting for a clock event in the `in_action` method, events are lost (signal values overwritten) or duplicated (signal values read twice) from the asynchronous data channel. Thus, the correctness of this handshake protocol heavily depends on the relative speed of `module1` and `module2`.

12.4.3.3 Asynchronous Inter-Module Communication between Clock-Sensitive Modules

To resolve both previous problems, we consider a safe asynchronous inter-module communication channel for SystemC modules with clock-sensitive listening threads. An asynchronous protocol for GALS communication is the *four-phase asynchronous handshake*. This protocol applies back pressure when the sender sends data the receiver cannot accommodate. The protocol starts at an initial idle state. This state indicates that the sender is not ready to transmit and can be entered again during the protocol. The protocol defined at the network level uses a request (`s_ready`), an acknowledgement signal (`d_ready`), and one or more data signals. The sender asserts the request, and the receiver responds by asserting the acknowledgement signal. Then, both signals are de-asserted in sequence: first, the sender withdraws its request, followed by the receiver withdrawing the acknowledgement signal.

This fully interlocked asynchronous handshake allows correct operation without loss or duplicate information. We simplify things and focus (without loss of generality) on the channel implementation of two interacting modules: sender and receiver; however, the concept extends to larger configurations.



FIGURE 12.6 Sender and Receiver connections.

Module connections are as shown in Figure 12.6.

```
class sender: public sc_module {
        public:
        SC_HAS_PROCESS(device);
        sender(sc_module_name nm):sc_module(nm) {
              SC_THREAD(out_action);
                    sensitive << CLK;
        }
        sc_in_clk CLK;
        sc_in<bool> d_ready;
        sc_out<bool> s_ready;
        sc_out<data_type> data_port;
        void out_action();
};
class receiver: public sc_module {
        public:
```

```
        SC_HAS_PROCESS(device);
        receiver(sc_module_name nm):sc_module(nm) {
              SC_THREAD(in_action);
                    sensitive << CLK;
        }
        sc_in<data_type> data_port;
        sc_in_clk CLK;
        sc_in<bool> d_ready;
        sc_out<bool> s_ready;
        void in_action();
};
void sender::out_action(){
        while(1){
              wait();
              s_ready.write(false);
              if (!d_ready.read()){
                    data_port.write(module_data);
                    s_ready.write(true); // ready to send
                    wait(d_ready.value_changed_event());
              }
        }
}
void receiver:: in_action() {
        while(1){
              wait();
              s_ready.write(false);
              if (d_ready.read()){
                    module_data = data_port.read();
                    s_ready.write(true);
                    wait(d_ready.value_changed_event());
              }
        }
}
int sc_main (int argc, char* argv[]) {
        ...
sc_clock clock1("CLOCK1", ZZ, SC_NS, X.X, Y.Y, SC_NS);
sc_clock clock2("CLOCK1", ZZ, SC_NS, X.X, Y.Y, SC_NS);
        ...
        sender sender0("SENDER");
        sender0.data_port(data_signal);
        sender0.m_s_ready(signal1);
        sender0.m_d_ready(signal2);
        sender0.CLK(clock1);
        receiver receiver0("RECEIVER");
        receiver0.data_port(data_signal);
        receiver0.m_d_ready(signal1);
        receiver0.m_s_ready(signal2);
        receiver0.CLK(clock2);
        sc_start();
}
```

As shown in the SystemC code above, we use two extra signal ports of type `bool`. Each port has two ports: `s_ready` (request) output port and `d_ready` (acknowledgment) input port. The `s_ready` output port of the sender is connected to the `d_ready` listening port of the receiver, and vice versa. The above implementation works as follows.

When the receiver sets its `s_ready` output signal to false (ready to receive), the `!d_ready` input signal of the sender module becomes true, and consequently, the `out_action` thread in the sender module can send data to the receiver via its `data_out` port. Then, it sets its `s_ready` signal

to true and waits until its `d_ready` signal changes value, i.e., until it is true.

The change in the sender's `s_ready` signal allows the receiver module, which listens to the corresponding `d_ready` signal, to read data from its `data_port`. Subsequently, it sets its `s_ready` output signal to true and waits until its `d_ready` signal changes value, i.e., until it becomes true. The change in the `s_ready` signal of the receiver unblocks the sender who listens to its `d_ready` port via the `d_ready.value_changed_event()`.

Notice that both sender and receiver calls to the `value_changed_event()` are necessary to avoid duplicating/overwriting the transmitted data when the relative speeds of the sender and receiver differ. Moreover, by using Boolean signals, no messages are lost, resolving the issue discussed earlier. Finally, boolean (and bit-vector) signals are directly synthesized with HLS tools.

This SystemC implementation allows further back-annotation, e.g., by calling either a simple `wait(x, SC_NS)` for a given number of ns or possibly `wait(x, SC_NS, ready.value_changed_event())`, which waits for a specified time of `x` nanoseconds or event `ready` to occur, whichever comes first. However, semantics (earliest of the two) can lead to duplicates and require further handling. Moreover, a call to `wait(x, SC_NS)` (for an arbitrary `x`) could lead to synchronization hazards (deadlock, message loss, duplicates) since events generated at specific instances may never be consumed.

As an extension, the above handshake protocol can be rewritten based on a push instead of a pull functionality, i.e., the sender (instead of the receiver) initiates the transaction. This implementation, explained in [62], can be used to design multiaccess channels (e.g., an asynchronous router) using `sc_spawn()` and an appropriately designed arbiter.

12.4.4 Behavioral PLL Model in SystemC-AMS

In most embedded SoC devices, such as wireless radio, digital circuits are combined with analog circuits. Most analog systems have a small transistor count, such as operational amplifiers, data converters (ADC or DAC), filters, phase-locked loops (PLLs), fluidic and chemical/biological systems, mechanical accelerometers and electrical sensors, and power management chips. Complex analog blocks include analog controllers in production, e-Health biosensor analog front ends, transportation motor controllers, satellite power systems, and consumer electronic appliances, especially RF units [63] [64] [65] [66]. Analog mixed-signal (AMS) design must address design complexity, power efficiency, verification, and signal integrity at all levels, from single-cell devices to high-level components and from the discrete to the continuous time domain. Although digital design is highly automated, analog design has been described as *flat, non-hierarchical,* and *intertwined* with many component interactions. As a result, critical analog circuits require a disproportionately large amount of chip area and require (comparatively) immense effort in interface design, validation, manufacturing, analysis, and IP reuse.

In this context, system-level modeling, design space exploration, and virtual prototyping of embedded analog/mixed-signal systems may use an innovative SystemC-based approach.

The SystemC AMS 1.0 standard provided by Fraunhofer first enabled describing analog/mixed-signal behavior using three models of computation (Timed Data Flow (TDF), Linear Signal Flow (LSF), and Electrical Linear Networks (ELN)). In 2013, the SystemC AMS 2.0 standard proposed by Accellera, with reference implementation from Fraunhofer, supported simple and efficient modeling of discrete-event and continuous systems at an elevated level of abstraction. The latest revision (IEEE Std 1666.1-2016) supports functional modeling, refinement, exploration, and virtual prototyping

of complex AMS systems [67] [68]. COSEDA provides an open reference implementation based on SystemC standard, COSEDA also provides the COSIDE IDE [69] [70] supporting user-friendly design and verification of heterogeneous mixed-signal systems, including sensors, RF, automotive networks (e.g., LIN, CAN), SATA interfaces, and QEMU emulation.

The SystemC-AMS extension complements the SystemC language with circuit-level simulation for critical AMS or RF circuitry (like Spice tools) and behavioral modeling of analog/mixed-signal applications at higher levels of abstraction (like Verilog-AMS, SystemVerilog, or VHDL-AMS). SystemC-AMS provides a high-level abstraction to support fast simulation of next-generation complex multi-million transistor SoCs.

The SystemC-AMS standard defines a User Guide and Language Reference Manual that specify execution semantics and extensions to SystemC language constructs: classes, interfaces, analog kernel, and modeling of continuous-time analysis. It aims towards developing executable specifications of embedded AMS systems at the system-level, behavioral, and netlist levels in the same simulation environment. This extension made SystemC the first AMS language combining frequency and time domain analysis, enhancing system modeling and hardware/software co-design in complex applications, such as electro-mechanical system control in robotics and co-simulation of analog and digital blocks of wireless Bluetooth networks.

More explicitly, SystemC AMS extensions define new language namespace constructs identified by the prefix `sca_`. Depending on the underlying semantics, AMS modules can be declared in dedicated namespaces [67] [70]:

- `sca_tdf` (timed data flow semantics, which include modeling systems in which activation periods or frequencies are either statically defined or dynamically changing),
- `sca_eln` (electrical linear networks), and
- `sca_lsf` (linear signal flow).

Using namespaces, similar primitives to SystemC are defined to denote ports, interfaces, signals, and modules. For example, a timed data flow input port is an object of class `sca_tdf::sca_in<type>`.

12.4.4.1 SystemC-AMS Parameter Annotation

SystemC-AMS behavioral models focus on functionality rather than on the details of the physical implementation, enabling fast simulation and design space exploration early in the design phase, i.e., before focusing on implementation details of low-level electrical circuit models and process technology. In contrast, low-level models allow precise evaluation of parasitic effects and electrical parameters of the analog circuit, such as wire delay parameters. This low-level characterization allows back annotation of the initial behavioral models (at object instantiation time in SystemC-AMS) with several architecture-related characteristics: performance, power, and reliability for clock-approximate or clock-accurate modeling.

Then, a back-annotated behavioral model can be used to

- ensure correctness in the presence of hazards,
- perform accurate system-level design space exploration and analysis, and
- enable interfacing to new hardware logic or software models.

For example, analysis using parameter sweeps helps estimate the sensitivity of a system-level design to perturbations in the architecture in different operational environments and allows the examination of new features in derivative products.

12.4.4.2 PLL Definition in SystemC-AMS

As a case study, we consider using SystemC-AMS timed data flow for behavioral modeling and skew reduction of self-clocking phase-locked loop (PLL) [71] [72]. We explain how to annotate model parameters from existing open or freeware low-level electrical circuit designs to perform accurate system-level simulation, operational characterization, consistency validation, and sensitivity analysis. In addition, it is possible to accurately validate a PLL's performance, reliability, and signal-to-noise ratio concerning aging by modeling different types of noise, distortion, and interference effects.

For the operational characterization of the PLL, we can use behavioral-level simulation in SystemC-AMS with input/output timed data flow (`sca_tdf`) signals. Execution of a TDF model can be suspended when waiting for discrete events from SystemC modules or when signals are sampled or interpolated across the digital/analog domain through converter ports. The PLL module described in SystemC-AMS models all circuit blocks, i.e., phase detector, low pass filter (which can be implemented in the frequency domain using `sca_lsf`), and analog/digital voltage-controlled oscillator (VCO). The block will generate an output signal, which has a fixed (mathematically proven) relationship to the frequency of an input reference signal.

Moreover, the PLL can be integrated with external modules that introduce perturbations to parameters and input signals (e.g., the reference oscillator). These modules, also modeled as generic SystemC-AMS classes with input and output TDF signals, can cause variations due to nonlinear effects and thermal Gaussian noise [73] [74] [75], such as those resulting from component aging. Block specifications and bindings are omitted.

The PLL class definition in SystemC-AMS is as follows.

```
pll(sc_core::sc_module_name n,
       double ph_gain, // phase detector gain
       double lp_ampl, // low pass amplitude
       double lp_fc,   // low pass frequency
       double vco_fc,  // vco frequency
       double kvco,    // vco sensitivity (Hz/V)
       double vco_gain,// vco gain
       double vco_vc)  // vco voltage control
```

Its timed data flow signals at the I/O interface are:

```
sca_tdf::sca_in<double> ref;  // reference signal
sca_tdf::sca_out<double> lpo; // low pass filter
sca_tdf::sca_out<double> vco; // vco
```

The PLL model design parameters are as follows: (specification and binding details are similar to SystemC and are omitted)

Parameter	Default	Description
`sc_module_name n`	-	
`double ph_gain`	3.7	gain of phase detector
`double lp_ampl`	1	amplitude of low pass filter
`double lp_fc`	1	cutoff frequency of low pass filter (Hz) - small to remove high-frequency clock noise
`double vco_fc`	10^9	the central frequency of vco output (Hz)
`double kvco`	10^7	sensitivity of vco (Hz/V)

double vco_gain	1	gain of vco output
double vco_vc	0	voltage control of vco
integer rate	1	data rate of input port (no multi-rate channels)

Loop filter parameters (e.g., gain) and phase detector or VCO tuning characteristics affect PLL behavior. For example, abrupt small perturbations (e.g., ±1%) in the reference frequency or phase or filter dynamics (loop bandwidth, damping factor, and frequency) influence PLL operational characteristics and stability, e.g., lock time, cycle slipping and damping behavior, reference spurs and phase noise. For instance, a lower bandwidth filter causes the loop to respond slower to the noise injected by the VCO, while a higher bandwidth allows the loop to respond quickly to the noise and compensate for it.

12.4.4.3 PLL Parameter Annotation

For a realistic accurate and efficient simulation of PLL operational characteristics, the previously described behavioral-level SystemC-AMS models must be annotated with system state parameters from corresponding low-level electrical implementations. More specifically, it is necessary to modify SystemC-AMS model parameters, including initialization of constant system parameters (related to timing and power characteristics) and dynamic modification of parameter values during simulation, e.g., when modeling specific stimuli or perturbations due to component interactions or technology-related effects, such as noise processes.

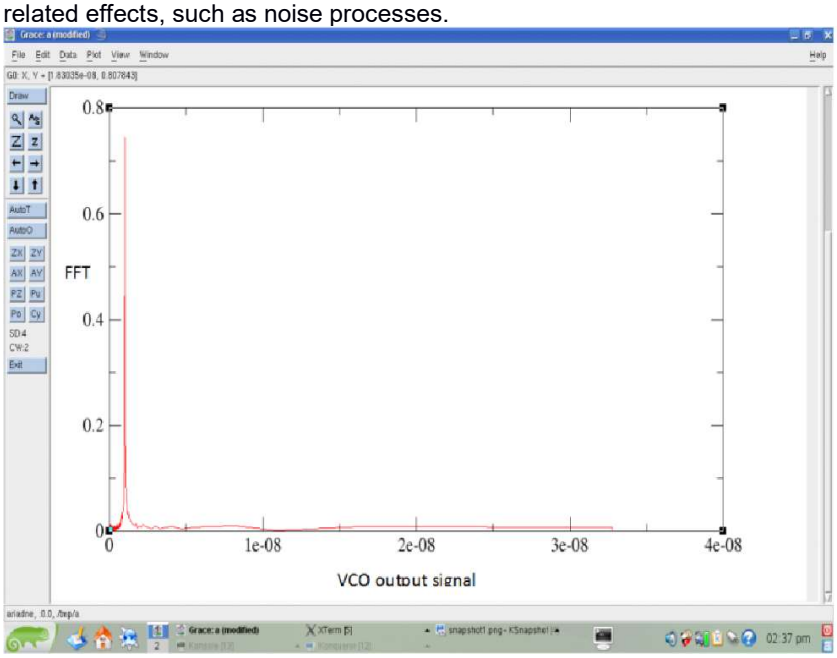


FIGURE 12.7 FFT of PLL VCO output signal.

Figure 12.7 shows a typical frequency analysis after simulation, which shows a peak at the required frequency of 1GHz. Spurs have much smaller amplitude by ~20dB and reduce PLL capture range and lock time [49,50]. For a refined analysis, we can compute peak-to-peak and rms phase jitter and perform spur reduction using more sophisticated filters [51,52].

Low-level implementations at the electrical network level and circuit level are available in commercial (e.g., Spectre AMS from Cadence, and LabView from National Instruments) and open PLL design toolkits [76] [77]. The latter toolkits support useful open-source C++ classes for analog mixed-signal

modeling, system performance metrics, and test bench scenarios. Unfortunately, since back-annotation from low-level AMS models is not seamlessly integrated with the SystemC-AMS modeling environment, circuit-level parameters can rarely be used without adaptation, a time-consuming and error-prone manual process even for the same process technology.

12.4.5 SystemC Security Annotation of Watchdog Timer

Security annotation differs from time annotation in digital circuit design and parameter animation in AMS design, which extracts the characteristics of the physical implementation to annotate high-level SystemC models for high-performance simulation and design space exploration. Instead, security annotation aims to extract security-sensitive data objects according to the IEEE P3164 standard by identifying security assets and concerns in a digital hardware module/subsystem design, using EDA tools or manually.

Security annotations are useful for security-related design space exploration during module-level design and system-level integration. However, their primary purpose is to define a threat model for security assurance and verification so that further hardware, firmware, or software development can achieve specific security objectives. The system-level integrator reviews the security weakness (APSO) objects included at the module level or the third-party IP Bundle to find which ones are in the scope of security concern. Some APSO objects may not be of any concern in a specific system-level integration. However, others may be relevant to module integration at the system level. In this case, the integrator should add the APSO objects to the threat model and examine possible mitigations.

The security annotation extracted from SystemC modules (subsystems) can be located outside the SystemC module itself. Hence, it can be applied to the existing design even if the hardware source (e.g., RTL) is encrypted. In this sense, different from time or power annotation in digital design and parameter annotation in AMS design, the security annotation is more similar to the automatic extraction of statistical information using enable/disable functions, as described in the following section.

12.4.5.1 Watchdog Module in SystemC

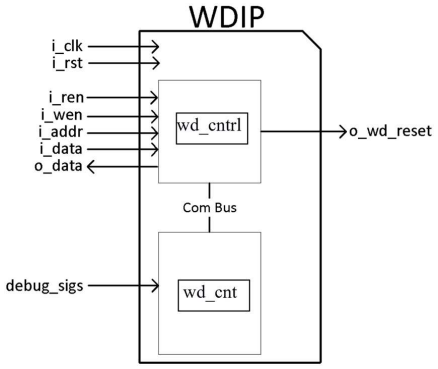


FIGURE 12.8 Watchdog Module: Block Diagram.

The Watchdog IP (WDIP) is a simple timer. When it expires, it asserts an output signal that can be used to put a function block into a known state. The timer counts down from an initial value that is the concatenation of the `REG_COUNT_HIGH` and `REG_COUNT_LOW` registers. The block diagram of the WDIP is shown in Figure 12.8.

The SystemC model of the above WDIP module is as follows.

```
#include "systemc.h"
SC_MODULE(wd_top)
```

```
{
    //Internal Signals...
    sc_in<bool>    i_clk;
    sc_in<bool>    i_rst;
    sc_in<bool>    i_ren;
    sc_in<bool>    i_wen;
    sc_in<sc_uint<8> >    i_addr;
    sc_in<sc_uint<8> >    i_data;
    sc_in<sc_uint<16> >    i_dbg_cnt_val;
    sc_in<bool>    i_dbg_enable;
    sc_in<bool>    i_dbg_clk_en;
    sc_in<bool>    i_dbg_clk;
    sc_in<bool>    i_dbg_timeout;

    sc_in<bool>    i_dbg_start;
    sc_in<bool>    i_dbg_service;
    sc_in<bool>    i_dbg_pause;

    sc_out<sc_uint<8> > o_data;
    sc_out<bool>    o_wd_reset;

    sc_signal<bool>    wd_timeout;
    sc_signal<sc_uint<16> >    timer_status;
    sc_signal<sc_uint<16> >    count_val;

    sc_signal<bool>    wd_start;
    sc_signal<bool>    wd_service;
    sc_signal<bool>    wd_pause;

    //Instantiated Modules Object Declaration...
    wd_count      count_block;
    wd_control    control_block;

    //Constructor Declaration...
    SC_CTOR(wd_top) :
        count_block("COUNT_BLOCK") ,
        control_block("CONTROL_BLOCK")
    {
        count_block.clk(i_clk);
        count_block.rst_n(i_rst);
        count_block.wd_timer(timer_status);
        count_block.wd_timeout(wd_timeout);
        count_block.count_val(count_val);
        //…
        control_block.clk(i_clk);
        control_block.i_wd_rst(i_rst);
        control_block.timer_status(timer_status);
        control_block.wd_timeout(wd_timeout);
        control_block.count_val(count_val);

        control_block.wd_start(wd_start);
        control_block.wd_service(wd_service);
        control_block.wd_pause(wd_pause);

        control_block.reg_rd_enb(i_ren);
        control_block.reg_wr_enb(i_wen);
        control_block.i_reg_address(i_addr);
        control_block.i_reg_data(i_data);

        control_block.o_wd_reset(o_wd_reset);
```

```
        control_block.o_reg_data(o_data);
    }
};
```

The top module consists of two basic blocks, *wd_cntrl* and *wd_cnt*. The *wd_cntrl* is used to configure the watchdog settings, including enabling, disabling, and servicing the timer. The *wd_cnt* block is the actual timer of the watchdog and communicates to the *wd_cntrl* block through the parallel "Com Bus". The "Com Bus" is internal to the watchdog IP and includes:

- *count_val[15:0]*: the initial value of the timer,
- *wd_timer[15:0]*: the current timer value,
- *wd_start*: to start the timer,
- *wd_service*: to service the timer (i.e., reset the count),
- *wd_pause*: to pause the timer, and
- *wd_timout*: Counter has reached zero.

The SystemC source code for the *wd_cnt* and *wd_cntrl* is as follows:

```
SC_MODULE(wd_count)
{
 //Internal Signals...
    sc_in<bool>    clk;
    sc_in<bool>    rst_n;
    sc_out<sc_uint<16> >    wd_timer;
    //…
    //Process Declarations
    void assign_process_wd_clk();
    void assign_process_wd_start();
    void assign_process_wd_service();
    void assign_process_wd_pause();
    void assign_process_count_val();
    void assign_process_wd_timeout_31();
    void always_process ();
    void always_process_54();

    //Constructor Declaration...
    SC_CTOR(wd_count)
    {
        SC_METHOD(assign_process_wd_clk);
        sensitive << i_dbg_enable << i_dbg_clk_en
<< i_dbg_clk << clk;

        SC_METHOD(assign_process_wd_start);
        sensitive << i_dbg_enable << i_dbg_start
<< wd_start;
        //…
        SC_METHOD(always_process);
        sensitive << wd_clk_w.posedge_event();
    }
};

SC_MODULE(wd_control)
{
    //Internal Signals...
    sc_in<bool>    clk;
    sc_in<bool>    i_wd_rst;
    sc_in<sc_uint<8> >    i_reg_address;
    //…

    //Processes Declaration...
```

```
    void assign_process_o_wd_reset();
    void assign_process_lock_flag();
    void assign_process_start_flag();
    void assign_process_pause_flag();
    void assign_process_wd_start();
    void assign_process_wd_pause ();
    void assign_process_service_flag();
    void assign_process_wd_service();
    void assign_process_o_reg_data();
    void assign_process_reg_write();
    void assign_process_reg_read();
    void always_process();

    //Constructor Declaration...
    SC_CTOR(wd_control)
    {
        SC_METHOD(assign_process_o_wd_reset);
        sensitive << wd_timeout;

        SC_METHOD(assign_process_lock_flag);
        sensitive << reg_control;
        //…    …
        SC_METHOD(always_process);
        sensitive << clk.pos();
    }
};
```

12.4.5.2 Security Annotation Flow in the Watchdog Timer

Using the WDIP as an example, the EDA tool's security annotation methodology is as follows:

- Identify a database of known security weaknesses. In this example, the CWE database is used [49] [50], and the Database data object class type will be defined. For example,

```
{
  "ID": "CWE VIEW: Hardware Design",
  "Description": "A community-developed list of hardware weakness
types",
  "URI": "https://cwe.mitre.org/data/definitions/1194.html",
  "Version": "4.3"
}
```

- Identify the security asset(s). Inside the counter block, the *wd_timer* parameter holds the timeout value of the watchdog. The watchdog functionality may be used to detect an undesirable condition in the system. Therefore, an adversary might want to prevent this timeout from happening and may want to modify the counter (e.g., increase or reset its value). This makes the *wd_timer* an asset of the WDIP module. Similarly, once a timeout occurs, the indication (i.e., system reset) must be propagated to the top module without modification. This timeout is identified in the counter block as *wd_assert_timeout*. An adversary who gains control of or influence over this register can modify the behavior of the WDIP (e.g., block the assertion of output signal *o_wd_reset* or assert constantly to create a denial of service). Therefore, *wd_assert_timeout* is critical for proper operation, which makes it an asset. The definition for the asset object is as follows:

```
{
  "Name" : " wd_top.count_block.wd_count.wd_assert_timeout",
  "Description" : "Timeout assertion signal. Critical for proper
operation",
  "Family" : ["Counter/Timer","Test/Debug"],
  "Type" : ["Control", "Critical"],
  "Database_ID" : ["CWE VIEW: Hardware Design"]
}
```

- Generate the corresponding Element data objects. For the WDIP, there are four objects generated: two are associated with the *wd_timer* asset, and another two are associated with the *wd_assert_timeout* asset. The first Element data object is as follows.

```
{
  "Asset Name": "wd_top.count_block.wd_count.wd_timer",
  "Direction": "Input",
  "Security Weakness Reference": ["CWE-1244","CWE-1191","CWE-
1234"],
  "Ports": [
       "wd_top.i_clk",
       "wd_top.i_rst",
       "wd_top.i_ren",
       "wd_top.i_wen",
       "wd_top.i_addr",
       "wd_top.i_data",
       "wd_top.i_dbg_cnt_val" ],
       "wd_top.i_dbg_enable",
       "wd_top.i_dbg_clk_en",
       "wd_top.i_dbg_clk",
       "wd_top.i_dbg_timeout",
       "wd_top.i_dbg_start",
       "wd_top.i_dbg_service",
       "wd_top.i_dbg_pause",
      "Parameters" : ["wd_top.COUNT_SIZE"]
}
```

- There are four APSO data objects corresponding to the four Element data objects. The first two APSO data objects are related to *wd_timer* asset, the Integrity security objective needs to be upheld since this is what an adversary would want to alter. The first APSO is to protect the *integrity of the timer value*, the IP provides a locking mechanism. Only when the lock is not asserted, can the timer be manipulated, which is captured in the *Condition* attribute. The APSO data objects for *wd_timer* are as follows:

```
{
  "Name": "SO_1",
  "Asset Name": "wd_top.count_block.wd_count.wd_timer",
  "Security Objective" : "Integrity",
  "Description": "If the lock bit is not enabled then the counter
can be altered",
  "Condition": "(wd_top.i_wen=1) && (wd_top.i_addr=REG_CONTROL)
&& (wd_top.i_data[0]=0)",
  "Security Weakness Reference" : ["CWE-1244","CWE-1191","CWE-
1234"],
  "Attack Points": [
       "wd_top.i_wd_rst",
       "wd_top.i_wd_clk",
       "wd_top.i_enb",
       "wd_top.i_wen",
       "wd_top.i_addr",
       "wd_top.i_data"],
  "Parameters": ["wd_top.COUNT_SIZE"]
}
```

APSO object “SO_2” requires the *Condition* of debug mode to be enabled to violate the security objective.

```
{
  "Name": "SO_2",
  "Asset Name": "wd_top.count_block.wd_count.wd_timer",
  "Security Objective": "Integrity",
  "Description": "Debug signals can alter the counter",
  "Condition": "wd_top.i_dbg_enable == 1",
  "Security Weakness Reference": ["CWE-1244","CWE-1191","CWE-
1234"],
  "Attack Points": [
       "wd_top.i_dbg_enable",
       "wd_top.i_dbg_clk_en",
       "wd_top.i_dbg_clk",
       "wd_top.i_dbg_pause",
       "wd_top.i_dbg_start",
       "wd_top.i_dbg_cnt_val"],
  "Parameters": ["wd_top.COUNT_SIZE"]
}
```

- The last step is security annotation packaging to create a complete IP Bundle to release to the system level or the external party. This will include the source code, netlist and test benches, and the SA-EDI data

objects produced above. The SA-EDI data objects can be organized into JSON objects [48]; this schema may require assimilation for packaging using IP-XACT.

After the complete set of security annotations is imported into the EDA tools, the corresponding Element objects can be regenerated for verification and assurance. After that, the system integrator reviews the APSO objects to find which ones are in scope for the integrated subsystem (module). For example, the APSO object labeled "SO_2" may not be a concern if the debug ports are disabled in the IC. However, if the debug ports are to be connected, this object would be in scope since it provides a backdoor. The final APSO objects in the top-level design can help create the threat model for security verification and design space exploration.

12.5 Automatic Extraction of Statistical Info

System-level modeling is an essential ingredient of design flow. Data and control flow abstractions of system hardware and software components express functionality and performance characteristics necessary to identify bottlenecks. For example, virtual channels are utilized to avoid network and protocol deadlock, improve performance, and provide quality of service. While for software components, it is usually the responsibility of the user or the operating system to provide relevant key performance indicators, for hardware components and interfaces, it is essential to develop a statistical package that enables transparent internal access to the modeling objects. The generated statistical data may be analyzed on the fly using open-source visualization software tools, such as gtkwave, Gnuplot, Grace [78], or dumped to a file for subsequent data processing, e.g., via a spreadsheet or a specialized text editor.

12.5.1 Statistical Classes for System-Level Models

Dynamic performance metrics, such as latency, throughput, packet loss, resource utilization, or power consumption, are usually required from many hardware system modeling objects:

- *Inter-module communication and synchronization objects* (Message) that represent communication channel performance metrics for throughput, latency, or packet loss; and
- *Intra-module passive memory objects* (`Register`, `FIFO`, `LIFO`, `circular FIFO`, `Memory`, `Cache`) that reflect memory performance metrics for throughput, latency, packet loss, buffer size, and hit ratio. Similar metrics are possible for joint intra-module communication and synchronization channels, e.g., `Mailboxes`. However, these objects may be implemented in hardware in different ways, e.g., using control logic, and static memory, local or distributed.

Assuming a straightforward 2-D graphical representation, a statistical API for the above metrics can use the function `enable_stat(args)` that activates the previously instantiated data-capturing activity. Its arguments specify a distinct name of the modeling object, the absolute start and end time for statistics collection, the title and legends for the `x` and `y` axes, the time window for windowed statistics, i.e., the number of consecutive points averaged for generating a single statistical point, and a Boolean flag for stopping or restarting statistics during simulation.

Since complex systems involve time- (instant) and event-driven (duration) statistics, we provide two general monitoring classes, collecting instant and duration metrics from system components with the following functionality.

In *time-driven simulation*, signals usually have instantaneous values. These values can be captured at a specific time during simulation by calling the function:

- `stat_write (double time, double value).`

In *event-driven simulation*, recorded statistics for events must include arrival and departure time (or duration). Since the departure time is known later, the interface uses two functions: `stat_event_start` and `stat_event_end`. Thus, first, the user invokes the operation:

- `int a = stat_event_start(double arrival_time)`

to record the arrival time (e.g., in clock cycles) and save the unique location of the event within the internal table of values in a local variable `a`. Then, when the event's departure time is known, this time is also recorded at the correct location within an internal table of values by calling the stat_event_end function with the appropriate departure time.

- `void stat_event_end(double departure_time, a).`

Sometimes, the `stat_event_start/end` operations take into account memory addresses to compute duration statistics for burst operations, e.g., consecutive read/write accesses (or enqueue/dequeue) operations corresponding to the same memory block (or packet) for a `Register`, `FIFO`, `Memory`, `Cache`, or inter-module communication object, i.e., `Message`. Thus, a modified `StatDurationLib` class performs the basic `stat_event_start/end` operations transparently to the system model, using internal pointers to library objects. Then, upon arrival of an event, we invoke the command:

- `void stat_event_start(long int MAddr, double arrival_time)`

to record time in the next available location within the internal table by providing the current memory address (`MAddr`), the arrival time (`arrival_time`), and an undefined (-1) departure time.

Finally, upon a corresponding departure event, we invoke the command

- `void stat_event_end(long int MAddr, double departure_time)`

to search for the current `MAddr` in the internal table and update the entry corresponding to the defined departure time.

For duration statistics in 2-D graphic form, the y-axis point may indicate the time of occurrence of a read operation performed on the same address as a previous write operation, whose time of occurrence is the corresponding point on the x-axis. Then, event duration, representing the latency of data accesses from the same memory block, can be obtained by subtracting these two values. Using Grace, this can be derived efficiently using the `Edit data sets: create_new` (`using Formula`) option. Furthermore, note that data is first written and then read; inverse access, i.e., read before write, causes an error.

As explained before, statistics collection may either be performed by the user manually, or directly instantiated by the hardware modeling objects and enabled by the system model using library-internal object pointers. In the latter case, software probes are inserted into the source code of library routines using monitoring segments, which automatically compile the necessary probes. Software probes share resources with the system model, thus offering low cost, simplicity, flexibility, portability, and precise application performance measurement in a timely, frictionless manner.

Using the proposed statistical functions `enable_stat(args)`, we can derive application-specific statistics for many system modeling objects:

- *Average or cumulative average throughput over a specified time window* of `Register`, `FIFO` objects, `Memory`, `Cache`, and inter-module communication channel objects.
- *The instant, average, or cumulative average size of counter-based objects*, such as `FIFO`, LIFO, and `circular FIFO` objects; these classes estimate `Cache` hit ratio statistics (for read/write accesses) and packet loss probabilities.

- *Latency statistics* for read/write (or enqueue/dequeue) on `Register`, `FIFO`, `Memory`, `Cache`, and inter-module communication channels.

In addition to the previously described classes, which cover simple cases, it is sometimes necessary to combine statistical data from different modeling objects, e.g., from hierarchical memory units, to compare average read vs. write access times or compute cell loss in network layer communication. For this reason, we need new user-defined *joint or merged statistics* built on top of time- and event-driven statistics. Specialized parameters, e.g., Boolean flags for these joint statistical classes can lead to more detailed metrics.

We have discussed the automatic extraction of statistical properties from hardware modeling objects, but this approach can be extended to software modeling. For example, the `enable_stat_` construct for obtaining write throughput statistics (read throughput is similar) from a software FIFO object `buffer` is listed below. This call will generate a graph similar to Figure 12.9.

```
// enable stats in [0,500] with time window = 1 (sample)
enable_stat_throughput_read("buffer", 0, 500, 1,
  "Simulation Time", "Average Throughput (Writes)");
```

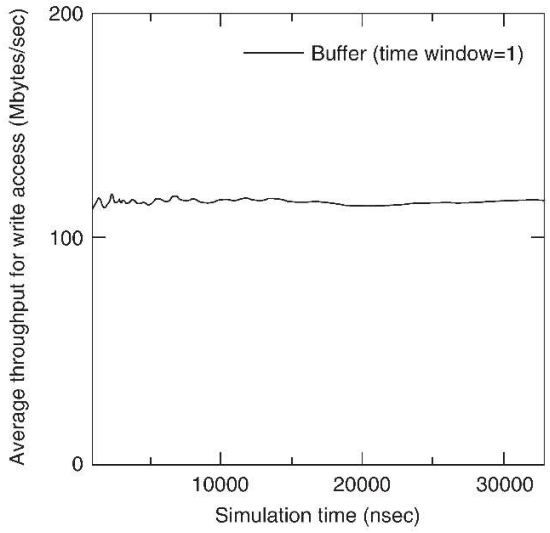


FIGURE 12.9 Performance results using Grace for a transport layer protocol.

For more details on the implementation of a statistical interface that supports multiple online or offline Grace-based statistical graphs, the reader is referred to the user manual and the statistical test benches accompanying the OCCN and HSoC libraries [15] [79]. These test benches are compatible with the GNU/Linux operating system.

12.5.2 Advanced System Monitoring

Conventional text output and source-code debugging are inadequate for monitoring and debugging complex and inherently parallel system models. System-level design tools generate VCD files for signal tracing or build relational database tables for data recording, analysis, and visualization. Efficient high-level performance modeling uses advanced monitoring activities.

- Generation refers to detecting events in non-intrusive ways and providing status and event reports containing monitoring traces (or histories) of system activity. Intrusiveness refers to side effects from monitoring or diagnostics on the monitored system due to sharing resources, e.g., processing power, communication channels, and

storage space.

- Processing monitoring data refers to merging and filtering traces, analysis and correlation, cross-validation, and updating. These functions must convert low-level monitoring info to the required format and level of detail.
- Presentation refers to displaying monitoring information in an appropriate form that provides visibility of interactions. This comprises textual information, animated time-process diagrams in synchronous systems, and concurrency maps in semi-synchronous and asynchronous event-based systems.
- Dissemination concerns the distribution of selected monitoring reports to system-level developers and external processing entities using push/pull schemes.

12.6 Open Issues in System-Level Modeling

Clock-accurate transactional system-level modeling is performed at an abstraction level higher than traditional RTL, thus improving design flow, enhancing design quality by offering efficient simulation, and system-level reuse. We have described efforts toward a general, rich, and efficient system-level design methodology that allows multiple levels of abstraction, enables hardware/software co-design, performance/power estimation, and verification, and provides interoperability with many state-of-the-art tools. Activities that could enhance this methodology include a variety of issues.

System safety, fault-tolerance, and validation/verification tools. SystemC tools exist only for verification [80]; these libraries are external to the system-level modeling library. SystemC Verification (SCV) provides a set of APIs for manipulating and sharing data objects across threads, generating randomness with constraints and reproducibility, and supporting transaction recording and temporal assertions for functional verification. Universal Verification Methodology (UVM) facilitates seamless integration/exchange of verification IP and test benches between system- and IP-level, i.e., SystemC/SystemC-AMS designs and SystemVerilog. Co-verification relies a) on constrained randomization using the CRAVE library for BDD-based and SAT/SMT-based constraint solvers and b) type- or instance-based functional coverage using the FC4SC library, which supports test creation, cover groups, bins, and cover points and exports coverage results using the Unified Coverage Interoperability Standard (UCIS) format. Note also that improving test coverage during validation requires transformations to verification languages [81] [82] or developing an asynchronous SystemC kernel scheduler [83] [84], which also results in different views of execution events from various objects, breaking sequential consistency.

AI and LLM techniques for efficient SystemC to-RTL synthesis [85]. New AI-enabled synthesis tools [86] [87] can reduce time-to-market for providing area, speed, and latency tradeoffs for complex designs, e.g., in signal and image processing. Unlike in the past, automated design of virtual platforms, which integrate analog mixed-signal should be considered part of the system-level design flow.

Parallelizing the SystemC kernel to execute efficiently in clusters or parallel systems enables efficient large-scale system-level simulation. Despite recent efforts to realize parallel SystemC simulations examined in [88] [59], most achievements are limited to thread- and data-parallel IP designs but not inherent SystemC/TLM language parallelism. This is due to synchronization overheads, event scheduling, shared resources, and race conditions that reduce the efficiency of parallel execution. Recent efforts from Busnot et al. provide a new parallel, standard-compliant SystemC kernel for loosely timed transaction-level models in bare-metal and Linux-based systems [90]. While approximately timed or cycle-accurate TLM simulations are more complex to

parallelize, loosely timed TLM models allow for greater flexibility in timing and are better candidates for parallelization. Another approach for loosely timed systems is to provide a set of primitives for the user to express tasks with long duration and limited interaction (`sc_during` pragmas) and let the system parallelize those tasks [91]. Finally, the SCale parallelization technique relies on efficient monitoring `sc_thread` accesses to shared resources with process-based rollback, if necessary [92]. This method requires atomic evaluation of all `sc_thread` processes to maintain the expected model behavior and ensure reproducible simulation. In addition, note that SystemC-AMS models are partitioned into clusters of highly interconnected modules with few dependencies and limited contention (e.g., multiple writers). Therefore, there is potential to achieve high parallelism by simulating SystemC-AMS clusters independently, requiring synchronization only at the borders. Despite being a promising development, this direction has not been explored yet.

Development of *full system prototypes, supporting GNU/Linux and safety-critical RTOS models.* The platforms must integrate software stacks, drivers, libraries, tools, and interfaces of embedded systems [18]. It is also important to consider *interoperability with system simulation tools*, and *system-level power dissipation or power intent* models from custom simulators [93].

In addition, there are many interesting open issues related to security annotation in high-level synthesis and integration [94]. Due to the unique properties of security annotation, a solid IEEE P3164 design flow has not been established yet in available EDA tools. Although security annotation can be completed manually outside the module/block design, automation of open and interoperable design toolsets is critical for preventing hardware and firmware security exploits in complex SoC designs. More specifically, key stepstones in security annotation are:

- *Automated extraction of security assets* (in JSON format) by the EDA tools and integration with existing SoC development flows. Security intent can be identified by examining intuitive inputs, outputs, and processing functions in a module's hierarchical architecture and its synthesized netlist. Based on this preliminary JSON output, the designer can finalize the list of asset data objects, so that EDA tools can automatically generate the corresponding Element and APSO data objects. It may also be possible to derive guidelines with mitigation policies for secure synthesis.
- *Automated security verification tools* are interesting. These EDA tools can use APSO data objects as input to evaluate different threat models, define realistic mitigation protocols, and eventually design test benches (or models), which can help validate (resp. verify) the desired security characteristics of the system and provide verification status reports. System-level security verification tools can benefit from applying fast, automated dynamic information flow tracking techniques using static or dynamic analysis of abstract syntax trees in the equivalent SystemC objects, cf. fundamental methodology in [95], implementation issues in [96] [97], and related examples on LEON3 SoCRocket VP and RISC-V VP-based SoC in [98] [99].
- System-level verification tools can be extended to support *reliable and secure programming patterns* during software development of the device driver, system, and application software. This is a challenge for the SystemC ecosystem, as it would make system-level design the backbone of a secure-by-design methodology that applies to hardware-software co-design and end-user applications.
- Finally, it is important to visualize system attributes, including security properties, for understanding complex IP [100]. This often requires configuration, control, and inspection (CCI) interfaces [systemcref] that instrument SystemC models with powerful debug,

monitoring, and analysis capabilities, including waves, concurrency maps, and system snapshots [101].

Acknowledgments

<to be added later>

References

[1] Carloni, L.P. and Sangiovanni-Vincentelli, A.L., "Coping with latency in SoC design," *IEEE Micro*, **22 (5)**, 2002, pp. 24—35.

[2] SystemC. [Online]. Available: https://systemc.org/overview/systemc and Downloads https://www.accellera.org/downloads/standards/systemc

[3] Murti, K. "Specification languages: SystemC", *In: A. Chakrabarty (ed), Design Principles for Embedded Systems*, TCSN 1657, Springer, 2022, pp. 85—117.

[4] IP-XACT, IEEE Std 1685. [Online]. Available: https://www.accellera.org/downloads/standards/ip-xact

[5] Krolikowski, S., Schirrmeister, F., Salefski, B., et al., "Methodology and technology for virtual component-driven hardware/software co-design on the system level," *in Proc. Int. Symp. on Circuits and Systems*, **6**, 1999, pp. 456–459.

[6] Synopsys, Synopsys Cocentric SystemC Compiler. [Online]. Available: https://www.inf.pucrs.br/~calazans/research/BrazilIP_project/SystemC_synth_prot_course/scc_bumg.pdf

[7] Ferrari, A. and Sangiovanni-Vincentelli, A., System design: traditional concepts and new paradigms, *in Proc. Int. Conf. Computer Design*, 1999, pp. 2–13.

[8] Rowson, J.A. and Sangiovanni-Vincentelli, A.L., Interface-based design, *in Proc. Design Automation Conf.*, 1997, pp. 178–183.

[9] Moriconi, M., Qian, X., and Riemenschneider, R.A., Correct architecture refinement, *IEEE Trans. Software Eng.*, **SE-21**, 1995, pp. 356–372.

[10] Medvidovic, N. and Taylor, R.N., A framework for classifying and comparing architecture description languages, *in Proc. Software Engin. Conf.*, LNCS1301, Spinger, Berlin, 1997, pp. 60–76.

[11] Charest, L., Aboulhamid, E.M., and Tsikhanovich, A., "Designing with SystemC: multi-paradigm modeling and simulation performance evaluation," *in Proc. HDL Conf.,* 2001, pp. 33–45.

[12] Virtanen, S., Truscan, D., and Lilius, J., "SystemC based object-oriented system design", *in Proc. Forum on Design Lang.*, 2001, pp. 1–4.

[13] Keutzer, K., Newton A. R., Rabaey J. M., et al., "System-level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, 2000, pp. 1523—1543.

[14] Brunel, J.-Y., Kruijtzer, W.M., Kenter, H.J.H.N., et al., "Cosy communication IP's," *in Proc. Design Automation Conf. (DAC)*, 2000, pp. 406—409.

[15] Coppola, M., Curaba, S., Grammatikakis, et al., OCCN: a NoC modeling framework for design exploration, *J. Syst. Arch. — Special Issue on Network-on-Chip*, Vol. 50, Elsevier, North Holland, 2004, pp. 129–163.

[16] Jiang, N., Becker D.U., Michelogiannakis, G., et al., "A detailed and flexible cycle-accurate network-on-chip simulator," in Proc. IEEE Int. Symp. Perf. Analysis Syst. & Soft. (ISPASS), 2013, pp. 86—96.

[17] Melkonian, M., "Get by without an RTOS," *Embedded Systems*, **13 (10)**, 2000.

[18] Lazzara, L., Mancuso, G.M., Cremona, F., et al., "A modular SystemC RTOS model for uncertainty analysis," *In: Chamberlain, R., Grimheden, M., Taha, W. (eds) Cyber Physical Systems: Model-Based Design*, LNCS 11971, 2019, pp. 3—27.

[19] ARM Cortex-A9 Technical Reference Manual (Appendix B: Cycle Timings). 2020, [Online]. Available: http://www.arm.com
[20] Heterogeneous SoC Library, available from http://hsoc.sourceforge.net
[21] Synopsys/Imperas OVP. [Online]. Available: http://www.ovpworld.org and http://www.ovpworld.org/dlp
[22] Carloni, L.P., McMillan, K.L., and Sangiovanni-Vincentelli, A.L., Theory of latency-insensitive design, *IEEE Trans. Computer-Aided Design of Integrated Circuits & Syst.*, **20–29**, 2001, pp. 1059—1076.
[23] Akesson, B., Goossens, K., and Ringhofer, M., "Predator: a predictable sdram memory controller," in Proc. IEEE/ACM Int. Conf. Hw/Sw codesign and system synthesis (CODES+ISSS), 2007, pp. 251—256.
[24] Benini, L., and De Micheli, G., "System-level power optimization: techniques and tools," ACM Trans. Design Automation Electr. Syst., 5 (2), 2000, pp. 115—192.
[25] Grammatikakis, M.D., Kornaros, G., and Coppola, M., "Power-aware multicore SoC and NoC design", *In: Hübner, M., Becker, J. (eds) Multiprocessor System-on-Chip*, Springer, 2010, pp. 167—193.
[26] Ye, T.T., de Michelli G., and Benini, L., "Analysis of power consumption on switch fabrics in network routers," *in Proc. ACM Design Automation Conf. (DAC)*, 2002, pp. 524—529.
[27] Yu, J., Wu, W., Chen, X., et al., "Assertion-based power/performance analysis of network processor architectures," *in Proc. Int. IEEE Workshop High-Level Design Validation and Test*, 2004, pp. 155—160.
[28] Unified Power Format (UPF), IEEE Std 1801, 2018, pp. 1—548.
[29] Unified Power Models (UPM), IEEE Std 2416, 2019, pp. 1—63.
[30] Greaves, D.J., and Yasin, M., "TLM POWER3: Power estimation methodology for SystemC TLM 2.0," *in Proc. Forum on Specification & Design Languages (FDL),* 2012, pp. 106—111.
[31] Vece, G.B., Conti, M., and Orcioni, S., "PKtool 2.0: a SystemC environment for high level power estimation," *in Proc. IEEE Int. Conf. Electr. Circ. Syst.*, 2005, pp. 1—4.
[32] Walravens, C., Vanderperren, Y., and Dehaene, W., "ActivaSC: a highly efficient and non-intrusive extension for activity-based analysis of SystemC models," *In Proc. ACM Design Automation Conf. (DAC)*, 2009, pp. 172—177.
[33] Qamar A., Muslim F.B., Iqbal J, et al., "LP-HLS: automatic power-intent generation for high-level synthesis-based hardware implementation flow", *Microprocessor and Microsystems*, **50**, 2017, pp. 26–38.
[34] Macko D., Jelemenska K., and Cicak P., "Simplifying low-power SoC top-down design using the system-level abstraction and the increased automation," *Integration*, **63**, 2018, pp. 101–114.
[35] Gagarski K.,, Petrov M., Moiseev M., et al., "Power specification, simulation, and verification of SystemC designs," in Proc. IEEE Symp. East-West Design & Test (EWDTS), 2016, pp. 1—4.
[36] Genov, A., Verdier, F., and Leconte, L., "Reusable system-level power-aware IP modeling approach design and verification", *in Proc. Design and Verification Conference Conf. (DVCON-US)*, 2022.
[37] Genov, A., Leconte, L., and Verdier, F., "Mixed electronic system level power/performance estimation using SystemC/TLM2.0 modeling and PwClkARCH Library", *in Proc. Design and Verification Conference and Exhibition (DVCON-Europe)*, 2020
[38] Ameur, A.B., Genov, A., Verdier, F., et al., "Timing-aware high level power estimation of industrial interconnect module,", *in Proc. Design and Verification Conference and Exhibition"(DVCON-Europe)*, 2020
[39] Chandrasekar, K, Weis, C., Li, Y., et al., "DRAMPower: Open-source DRAM power & energy estimation tool". [Online]. Available: http://www.drampower.info
[40] Bouhadiba, T., Moy, M., and Maraninchi F., "System-level modeling of energy in TLM for early validation of power and thermal management,"

in Proc. Design, Automation & Test in Europe Conf., 2013, pp. 1609—1614.
[41] Bouhadiba, T., Moy, M., Maraninchi, F., et al., "Co-simulation of functional SystemC TLM Models with power/thermal solvers," *in IEEE Int. Symp. Parallel & Distr. Processing (IPDP)*, 2013, pp. 2176—2181.
[42] Michaud, P., and Sazeides, Y., "ATMI: analytical model of temperature in microprocessors," *in Proc. Workshop on Modeling, Benchmarking and Simulation*, 2007.
[43] Huang, W., Member, S., Ghosh, S., et al., "Hotspot: A compact thermal modeling method for CMOS VLSI systems," *IEEE Transactions on VLSI Systems*, **14(5)**, 2006, pp. 501—513.
[44] Moy, M., Helmstetter, C., Bouhadiba, T., et al., "Modeling power consumption and temperature in TLM models," *Leibniz Transactions on Embedded Systems*, 2016, **3 (1)**, pp. 1—29.
[45] Verbauwhede, I., "Security adds an extra dimension to IC design: future IC design must focus on security in addition to low power and energy," *IEEE Solid State Circ. Magazine*, **9(4)**, 2017, pp. 41-45.
[46] Hu, W., Chang, C.-H., Sengupta, A., et al., "An overview of hardware security and trust: threats, countermeasures, and design tools," *in IEEE Trans. CAD Integr. Circ. & Syst.*, **40(6**), 2021, pp. 1010—1038.
[47] Grammatikakis, M.D., Papadimitriou, K., Petrakis, P., et al., "Security in MPSoCs: A NoC Firewall and an Evaluation Framework," in IEEE Trans. on CAD Integr. Circ. Syst., **34(8)**, 2015, pp. 1344—1357.
[48] Accellera, Security Annotation (SA_EDI). [Online]. Available: https://www.accellera.org/images/downloads/standards/Accellera_SA-EDI_Standard_v10.pdf
[49] Mitre CWE. [Online]. Available: https://cwe.mitre.org
[50] NIST CWE. [Online]. Available: https://nvd.nist.gov/vuln/categories
[51] Binkert, N., Beckmann, B., Black G., et al., "The gem5 simulator," *SIGARCH Comput. Archit. News,* **39 (2)**, 2011, pp. 1–7.
[52] Lowe-Power, J., Ahmad, A.M., Akram, A., "The gem5 simulator: Version 20.0+," 2020, Arxiv. https://arxiv.org/abs/2007.03152
[53] Menard, C., Castrillon, J., Jung, M., et al., "System simulation with gem5 and SystemC: the keystone for full interoperability," *in Proc. Int. Conf. Embedded Computer Systems - Architectures, Modeling, and Simulation (SAMOS)*, 2017, pp. 62—69.
[54] Fennibay, D., Yurdakul, A., and Sen, A., "A heterogeneous simulation and modeling framework for automation systems", *in IEEE Trans. CAD Integr. Circ. & Syst.*, , **31 (11)**, 2012, pp. 1642—1655.
[55] Papagrigoriou, A., Grammatikakis M.D., and Piperaki, V., "A hybrid channel for co-Simulation of behavioral systemC IP with its full system prototype on FPGA," *in Proc. Design and Verification Conference and Exhibition (DVCON-Europe)"*, 2013, pp. 1609—1614.
[56] Gutierrez, A., Pusdersris, J., Dreslinksi, T., et al., Source of error in full-system simulation, *in Proc. Int. Symp. Performance Analysis Systems and Software*, 2014, pp. 13–22.
[57] Herdt, V., Drechsler, R., "Advanced virtual prototyping for cyber-physical systems using RISC-V: implementation, verification, and challenges," *Science China Information Science*, **65(1)**, 2022,pp. 1—17.
[58] Monton, M., "A RISC-V SystemC-TLM simulator," in Proc. Workshop Comp. Arch. Research with RISC-V (CARRV), 2020. Code/testbenches [Online]. Available: https://github.com/mariusmm/RISC-V-TLM.
[59] Harel, D., Statecharts: a visual formalism for complex systems, *Sci. Comput. Program.*, **8**, 1987, pp. 231–274.
[60] von der Beeck, M., "A comparison of statecharts variants," *In Proc. Formal Techn. in Real Time and Fault Tolerant Syst.*, LNCS 863, Spinger, 1994, pp. 128–148.
[61] Sheibanyrad A., Greiner A., and Panades I.M., "Multisynchronous and fully asynchronous NoCs for GALS architectures," *IEEE Design and Test*

of Comp., **25 (6)**, 2008, pp. 572–580.
[62] Crews M., and Yuenyongsgool Y., “Practical design for transferring signals between clock domains,” *EDN Network*, **48 (4)**, Feb. 2003, pp. 65–60.
[63] Fraccaroli, E. and Vinco, S., "Modeling cyber-physical production systems with SystemC-AMS," *IEEE Transactions on Computers*, **72(7)**, 2023, pp. 2039—2051.
[64] Haeseker, J.S., Aksteiner, N., and Ofenloch, A., “From behavioural to in-depth modelling of a scalable spacecraft power system in SystemC-AMS using continuous integration techniques,”, *Trans. Japan Society for Aeronautical and Space Sciences Aerospace Tech.*, **19-5**, 2021, pp. 709—718.
[65] Chen, Y., Baek, D., Kim, J. et al., "A SystemC-AMS framework for the design and simulation of energy management in electric vehicles," *IEEE Access*, **7**, 2019, pp. 25779—25791.
[66] Zimmermann, T., Mora, M., Steinhorst, S., et al., “Analysis of dissipative losses in modular reconfigurable energy storage systems using SystemC TLM and SystemC-AMS,”, *ACM Trans. Des. Autom. Electron. Syst.,* **24**, 2019, pp. 1—33.
[67] SystemC-AMS, IEEE Std 1666.1 standard, 2016. [Online]. Available: https://standards.ieee.org/ieee/1666.1/5812
[68] Accelera, SystemC-AMS publications. [Online]. Available: https://systemc.org/overview/systemc-ams
[69] Vachoux A., Grimm C., and Einwich K., “Towards analog and mixed-signal SoC design with SystemC-AMS,” *in Proc. IEEE Workshop on Electronic Design, Test and Appl.*, 2004, pp. 97—102.
[70] Pêcheux, F., Grimm, C., Maehne, T., et al., "SystemC AMS based frameworks for virtual prototyping of heterogeneous systems," *in Proc. IEEE Int. Symp. Circ. and Syst. (ISCAS)*, 2018, pp. 1—4.
[71] Antao B., El-Turky F., and Leonowich R., Mixed-Mode Simulation of Phase-Locked Loops, *in Proc, IEEE Custom Integr. Circ. Conf.*, 1993, pp. 841—844.
[72] Best R., *Phase-Locked Loops: Design, Simulation and Applications*, 4th Edition, McGraw-Hill, 1999.
[73] Nallatamby J.-C., Prigent M., Camiade M., et al., “Phase noise in oscillators—Leeson formula revisited,”, *IEEE Trans. Microwave Theory and Techn.*, **51 (4)**, 2003, pp. 1386—1394.
[74] Razavi B., A study of phase noise in CMOS oscillators, *IEEE J. Solid State Circuits*, **31 (3)**, 1996, pp. 331—343.
[75] Chen, Y., Vinco, S., Macii, E., et al., "Fast thermal simulation using SystemC-AMS," *In Proc. Int. IEEE Great Lakes Symposium on VLSI (GLSVLSI)*, 2016, pp. 427—432.
[76] CPPSim, Perrot, M., MIT. [Online]. Available: http://www.cppsim.com
[77] EasyPLL Web, National Instruments. [Online]. Available: http://www.ti.com/tool/clockdesigntool
[78] Grace. [Online]. Available: http://plasma-gate.weizmann.ac.il/Grace
[79] Grammatikakis, M.D., Papagrigoriou, A., Petrakis, P., et al., "Monitoring-aware VP prototype of heterogeneous NoC-based multicore SoCs", *in Proc. Digital System Design Conf. (DSD)*, 2013, pp. 497—504.
[80] SystemC for Verification: SCV/UVM proposals, 2017. [Online]. Available: https://systemc.org/overview/systemc-verification/#systemc-verification-library-scv
[81] Herdt, V., Le, H. M., Grosse, D., et al., “Compiled symbolic simulation for SystemC," *in Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, 2016, pp. 1—8.
[82] Tasche, P., Monti, R.E., Drerup, S.E., et al., "Deductive Verification of Parameterized Embedded Systems Modeled in SystemC,". *In: Dimitrova, R., Lahav, O., Wolff, S. (eds) Verification, Model Checking, and Abstract Interpretation*, LNCS 14500, Springer, 2024, pp. 187—209.

[83] Mueller, W., Ruf, J., Hoffmann, D., Gerlach, et al., “The simulation semantics of SystemC,” *in Proc. Design, Automation & Test in Europe Conf. (DATE)*, 2001, pp. 64—71.
[84] Weinstock, J., Schumaher, C., Leupers, R., et al., “Time-decoupled parallel SystemC simulation,” *in Proc. Design Automation & Test in Europ Conf. (DATE),* 2014, pp. 1—4.
[85] SystemC synthesizable subset, version 1.4.7, 2016. [Online]. Available: https://www.accellera.org/images/downloads/standards/systemc/SystemC_Synthesis_Subset_1_4_7.pdf
[86] Chen, D., "Lightning talk: the next wave of high-level synthesis," *in Proc. ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1—3.
[87] Liao, Y., Adegbija, T., & Lysecky, R., “Are LLMs any good for high-level synthesis?.”, ArXiv. https://arxiv.org/abs/2408.10428
[88] Doemer, R., "Seven Obstacles in the Way of Standard-Compliant Parallel SystemC Simulation," *in IEEE Embedded Systems Letters*, **8(4)**, 2016, pp. 81—84.
[89] Liu, G., Schmidt, T., Cheng, Z., et al., "RISC Compiler and Simulator: Out-of-Order Parallel Simulatable SystemC Subset", *Univ. California at Irvine, Center for Emb. Cyber-Physical Systems, TR 19-04*, September 2019. [Online]. Available: https://www.cecs.uci.edu/~doemer/risc.html
[90] Becker, D., Moy, M, and Cornet, J., “ Parallel simulation of loosely timed SystemC/TLM programs: challenges raised by an industrial case study,” *Electronics*, **5(22)**, 2016, pp. 1—20.
[91] Moy, M., "Parallel programming with SystemC for loosely timed models: A non-intrusive approach," *in Proc. Design, Automation & Test in Europe Conf. (DATE)*, 2013, pp. 9—14.
[92] Busnot, G., Sassolas, T., Ventroux, N., et al., "Standard-compliant parallel SystemC simulation of loosely timed transaction level models: from bare-metal to Linux-based applications," *Integration*, 79, 2021, pp. 23—40.
[93] Li, S., Yang, Z., Reddy, D., et al., "DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator," *in IEEE Computer Architecture Letters*, 19 (2), 2020, pp. 106—109.
[94] Pundir, N., Farahmandi, F., and Tehranipoor, M., "Secure high-level synthesis: challenges and solutions," *in Proc. Int. Symp. Quality Electronic Design (ISQED)*, 2021, pp. 164—171.
[95] Hedin, D., and Sabelfeld, A., “A perspective on information-flow control,” Software Safety and Security, **33**, 2012, pp. 319—347.
[96] Goli, M., and Drechsler, R., "Early SoCs Information Flow Policies Validation Using SystemC-Based Virtual Prototypes at the ESL", ACM Trans. Embed. Comput. Syst., vol. 23, no. 5, Aug. 2024.
[97] Goli, M., and Drechsler, R., “VIP-VP: early validation of SoCs information flow policies using SystemC-based virtual prototypes”, *in Proc. Forum on Specification & Design Languages (FDL),* 2021, pp. 1—8.
[98] Pieper, P., Herdt, V., Grosse, D., et al., “Dynamic information flow tracking for embedded binaries using SystemC-based virtual prototypes”, *in Proc. IEEE/ACM Design Automation Conf. (DAC)*, 2020, pp. 1—6.
[99] Herdt, V., Grosse, D., Pieper, P., et al., “RISC-V based virtual prototype: an extensible and configurable platform for the system-level”, J. Syst. Arch., **109**, 2020, pp. 1—13.
[100] Goli M. and Drechsler, R., "Through the looking glass: automated design understanding of SystemC-Based VPs at the ESL," *in IEEE Trans. CAD Integr. Circ. Syst.*, **41 (4)**, 2022, pp. 1181-1185.
[101] Charest, L., Reid, M., Aboulhamid, E.M., and Bois, G., “A methodology for interfacing open source SystemC with a third-party software,” *in Proc. Design, Automation & Test in Europe Conf. (DATE)*, 2001, pp. 16–20.