Chapter 2

ESL Design and Verification

First, the state-of-the-art of modeling and verification at the ESL is briefly summarized in this chapter. As SystemC has been developed as a de-facto standard for system level design over the past few years, it is used for practical demonstration purposes in this book. Next, the introduced ESL methodologies and techniques are discussed under the SystemC perspective. Finally, our systematic debugging approach for ESL designs is described.

1 ESL DESIGN

The rising "design gap" demands for continuous improvements of the design productivity. One of the most critical issues is, how the available chip capacity, following Moore's Law, can be utilized by the chip designers. They have to develop even more complex integrated circuits and systems under fixed time-to-market and quality constraints. According to the ITRS report [ITRS07], a designer has a productivity of 125k gates per design-year using a state-of-the-art RTL-centered design and verification methodology. Here, designers describe an integrated circuit with an HDL (e.g. Verilog or VHDL) at RTL. There, the design is (almost) automatically synthesized down to circuit layout ready for fabrication. On the verification side, sophisticated verification tool suites are applied using techniques such as tracking and reporting information about the code coverage, or performing a constraint random simulation.

Now, we naively assume that a 10 million gates circuit shall be created from scratch. If the above mentioned design productivity of 125k gates is taken into account, 80 designers have to work for one year to complete the integrated circuit. Even more complex systems would require an unacceptable high number of designers or an improper long design time. Moreover, time-to-market, and rising cost and quality constraints demand for a new modeling and verification methodology.

Two basic approaches could help to improve the design productivity: The first one lifts the design level to a higher more abstract level above RTL. The second approach introduces a design reuse methodology by means of (third-party) IP components. Modern flows combine both approaches to maximize benefits.

F. Rogin and R. Drechsler, *Debugging at the Electronic System Level*,
DOI 10.1007/978-90-481-9255-7_2, © Springer Science+Business Media B.V. 2010

Designing above RTL is called ESL design. Generally, abstraction aims at the reduction of the effort to specify a desired functionality whereas unnecessary information is omitted. At the ESL designs are described by using higher level concepts such as communicating processes exchanging more abstract information. Here, the accurate timing or the parallelization of system functionality is not initially taken into account. In fact, the designer starts to specify the general function of the system and successively refines the specification until the concrete system, consisting of hardware and software, is fully implemented. Today, ESL design is supported by different SDLs where C/C++-based programming languages play a major role.

IP reuse is an important opportunity to improve the design productivity. The number of blocks to be defined from scratch becomes smaller if the designer can reuse existing IP components. IP blocks are taken either from former development projects or by using external IP providers. IP shall fulfill a number of requirements to enable a successful reuse. First, they should be of high-quality in terms of correctness and completeness. Second, the IP component interfaces have to be clearly defined and documented. Third, IP blocks should separate communication from behavioral parts to ease the integration into an existing design architecture.

1.1 ESL Design Flow

Currently, in the literature no universal design and verification flow for ESL design is documented. Rather, many different flow variants are stated. They are geared by customer and product requirements. The flow variants distinguish among each other by means of the introduced abstraction levels, use cases, and later application fields. The following section gives a short overview about important work that deal with the ESL design methodology. Baileys et al. [BMP07] give a comprehensive summary.

According to Gajski et al. [GV+94] the general ESL flow can be described by the three main steps: *specifying*, *exploring*, and *refining*. First of all, a functional specification of the desired system is developed. During the subsequent *exploration* phase, different design alternatives are compared in order to meet various requirements, e.g. performance, power consumption, or configurability. In the *refinement* phase more and more functional behavior is mapped onto a structural description consisting of hardware and software components. Subsequent refinement steps repeat exploration and refinement phases until the whole system is structurally specified. All other mentioned approaches adhere to this general procedure.

Kogel [Kog06] refers to use cases in terms of views to overcome the difficulties to specify an ESL flow at the level of abstractions. ESL modeling gears to many different domains, e.g. communication, time, structure, or

behavior. Each domain can be described at particular levels of abstraction, e.g. the communication behavior can be modeled as transactions, bus functional models, or pin-accurate descriptions. Kogel concentrates on the purpose of the model by defining use cases. The *Functional View* (FV) use case is intended to create an executable specification of the application. The *Architects View* (AV) use case targets to the architectural exploration of the developed system to meet the desired requirements such as performance, power, or maintainability. The *Programmers View* (PV) use case provides a virtual prototype platform which could be applied for early embedded software development in parallel to the hardware design. Finally, the *Verification View* (VV) use case targets at cycle-accurate system modeling. So, an accurate performance evaluation prior to the design implementation or a TLM-RTL co-verification can be performed.

Teich and Haubelt [TH07] introduce an approach which summarizes important abstraction levels and views. These levels and views are passed during embedded system design. This approach distinguishes between *architecture* and *logic* abstraction levels in case of hardware designs. The *block* and *module* levels describe abstraction levels in the software development domain. Beside the classification of system models according to their levels of abstraction, two orthogonal views are defined. The *behavioral* view takes only the functionality of the system into account while the *structural* view details the communication between hardware and software components.

The SystemC TLM2 standard [TLM2] does not focus on abstraction levels. In fact, the standard mentions particular use cases, such as software development, software performance analysis, or hardware architecture analysis. These use cases are supported by two different coding styles, i.e. loosely-timed, and approximately-timed. Coding styles guide the designer in model writing using particular programming interfaces.

Fujita et al. [FGP07] document a design flow for system level designs based on C/C++-based system level languages. Starting with a textual specification an executable specification, mostly in terms of a program, is created. The resulting *functional system model* describes the general application and is mainly used to explore different functional design alternatives. During the *architecture design* phase, also known as *hardware/software partitioning*, the designer decides which functionality is implemented either into dedicated hardware components or software programs. Software plays an important role to allow for reusability, i.e. easy reuse of software components in terms of IP, maintainability, i.e. easy bug fixing and feature extension in the application field, and flexibility, i.e. easy replacement of software components. The exact communication between partitioned components, including processing order and parallelism, is detailed in the subsequent *communication design* phase

resulting in the *communication model*. The software is usually implemented using a standard software development process. However, the functions that are assigned to hardware parts are further decomposed into structural units to take the target hardware platform into account. Finally, within the *implementation design* phase, the hardware parts of the communication model are synthesized into an RTL design using *high-level ESL synthesis techniques*. Several tools automate this step. Nevertheless, in practice it is usually still a manual or at most a semiautomatic step. Finally, the RTL design is implemented which comprises well automated steps such as logic or layout synthesis. The developed software descriptions can be directly extracted from the communication model and are compiled into executable machine code.

Bailey et al. [BMP07] published one of the first books that summarizes state-of-the-art in ESL design and verification methodology. One important contribution is the definition of a taxonomy for the ESL design space and the definition of common terms used in this domain. The proposed ESL taxonomy contains the five axes concurrency, communication, configurability, temporal, and data. This taxonomy spans a space for the definition and classification of ESL specification languages, design flows, and tools. Bailey et al. divides the ESL design flow into six main development steps: *specification and modeling*, *pre-partitioning analysis*, *partitioning*, *post-partitioning analysis and debug*, *post-partitioning verification*, and *HW/SW implementation*. In the specification and modeling step the designer translates the initial informal specification document successively into various executable models. During the pre-partitioning analysis step, the algorithmic design space is explored taking different constraints into account, e.g. time, space, power, complexity, or time-to-market. The partitioning process defines which parts of the functionality are implemented either in software or hardware. The effect of hardware/software partitioning is explored in the post-partitioning analysis and debug step. If necessary, a re-partitioning takes place to better meet the requirements. Then, the following post-partitioning verification step ensures that the originally intended behavior of partitioned software and hardware components is preserved. Finally, the HW/SW implementation step creates synthesizable RTL models as well as the productive software that shall run on the target hardware.

Summarized, all aforementioned work can be mapped more or less to the idealized ESL design flow sketched in Figure 2.1. Rather than this idealized and simplified linear top-down flow, a real flow is a mixture of bottom-up and top-down procedures. Moreover, the development of an integrated system consists of many refinement steps and backtracking paths that shall be not further detailed here. In reality, parts of the model are usually implemented at different abstraction levels at the same time. So, some parts could be specified

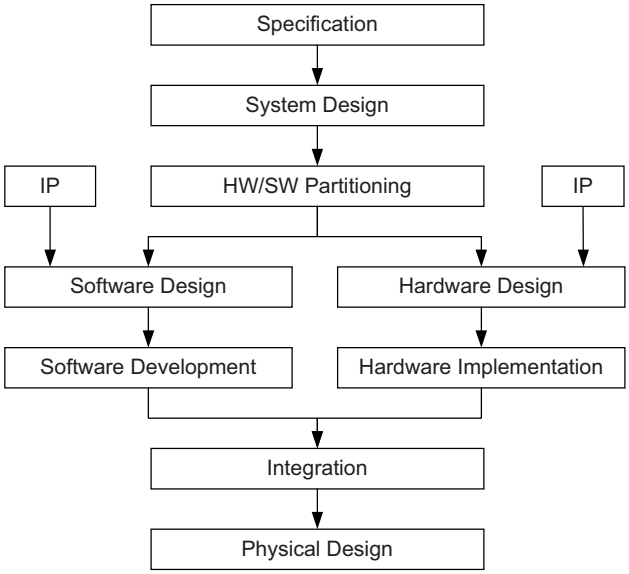

Figure 2.1: Idealized ESL design flow (taken from [BMP07])

at a more structural level while other parts are initially described at the behavioral level. The model can also contain loosely-timed specified components as well as cycle-accurate blocks in parallel. Nevertheless, a designed system always passes different levels of abstraction starting from the most abstract specification. Then, this specification is successively refined to the lowest most accurate level.

Figure 2.2 illustrates the design levels while implementing the hardware parts of a SoC design. It shows an idealized top-down procedure following a SystemC TLM oriented design methodology:

Starting with a textual specification, the *algorithmic design* is specified. Thus, the basic functionality of the developed system is explored while the distinction between hardware and software components as well as the timing is not yet taken into account.

Based upon the algorithmic specification, the design is refined to a TLM-based system model. The model consists of different blocks that are connected by communicating channels. Channels are used to exchange data between these blocks in terms of *transactions*. During development, the system is normally refined using different timing levels. At the *loosely-timed* level, the system model consists of functionally accurate components describing the system platform. Synchronization between components is only supported at a coarse-grained synchronization level specifying the general scheduling procedure, e.g. the correct order between produced and consumed data.

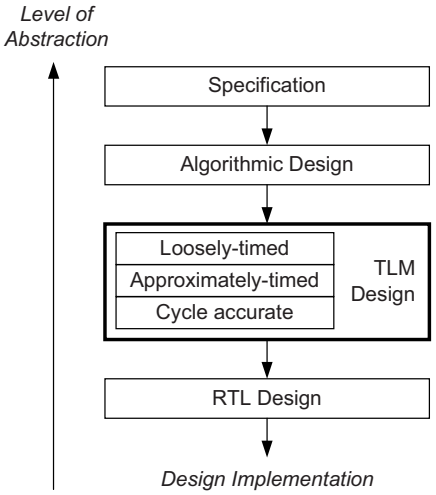

Figure 2.2: Designing the hardware part of an SoC

Moreover, communication channels are used for synchronization purposes, e.g. to handle interrupt signals. The loosely-timed abstraction level is the basis for early embedded software development and debugging where unnecessary implementation details are left out. The aforementioned PV use case relates to models at this level. At the next level, i.e. *approximately-timed*, the timing is refined which enables a preliminary performance estimation and a coarse power analysis. A model at this level is available earlier than the RTL model and can facilitate reasonable design decisions for the later RTL design. This abstraction level corresponds to the *PV plus Timing* use case. A *cycle-accurate* model describes the design behavior at the level of clock cycles. This abstraction level is used e.g. for final HW/SW partitioning, HW/SW co-verification, or a TLM-RTL co-simulation.

Transferring a system model into an RTL design is accompanied by a change of the design language. While system models are described using an SDL such as SystemC or SpecC, RTL designs are usually implemented using HDLs such as VHDL or Verilog. An RTL design contains all details of the hardware components and can be automatically and efficiently synthesized. Hardware synthesis comprises the design implementation, such as logic or layout synthesis, and is well supported by state-of-the-art tools and design flows. Detailed information to hardware design and synthesis can be found for instance in [Ash06], [Pal03].

1.2 System Level Language SystemC

C/C++-based languages have been playing a major role in embedded software development for a long time. So, it was apparent to use the same or some similar language to specify the hardware parts of an ESL design and to write test benches, as well. However, C/C++ do not provide constructs to deal with concurrency, communication, or timing necessary for designing hardware. Several extensions were proposed over the past few years to make C/C++ ready to be used in ESL design. Today, two C/C++-based SDLs are mainly used: SystemC and SpecC. Both languages provide similar language constructs for hardware design. The major difference is the underlying language. While SpecC bases upon C, SystemC is a C++ class library. Hence, SystemC has an object-oriented nature which facilitates the implementation of abstract, modular, and reusable system models. SpecC as well as SystemC are freely available which ease their application and distribution. Additionally, IEEE approved SystemC as a standard in 2005 [IE+06] which has increased the acceptance of this language in industry. The standardization process has been significantly driven by the OSCI. In this book, SystemC is used for demonstration purposes. However, the proposed debugging technique are applicable to any other system level language.

SystemC extends C++ by notions of modules, concurrent processes, simulation time, and communication mechanisms. These features support ESL design while comprising the full power of C++. A SystemC design is created by a hierarchy of (nested) modules which are instances of the class `sc_module`. Modules communicate through ports, interfaces, and channels. An interface provides several methods to access a channel whereas the methods are called through a port. A channel separates communication and computation. SystemC provides a number of predefined channel types ranging from simple wires to more complex communication mechanisms like FIFOs. Furthermore, designers can derive their own channel types. The main elements of computation are represented by concurrent processes that are either method (`SC_METHOD`) or thread processes (`SC_THREAD`). A process has to be specified inside a module. A thread process terminates never and suspends its computation by calling a `wait` statement. In contrast, a method process completely executes in zero time, every time it is triggered.

An integral part of the SystemC library is the event-driven simulation kernel which implements a cooperative multitasking approach and divides in two main phases:

- *Elaboration.* Execution starts at the `sc_main` function. First, all modules are instantiated and bound together. After this phase a fur-

ther change of the model structure is not possible. Then, the call of `sc_start` launches the simulation.

- *Simulation.* During simulation the SystemC kernel executes each runnable process one by one in a non-preemptive fashion. A process suspends again either when it was completely processed (method process) or the process calls a time consuming statement (thread process). There, the SystemC simulation kernel works like an HDL simulator using the notion of delta cycles. A delta cycle is orthogonal to the simulation time where the time is only advanced, when no more processes are runnable at the current time step, i.e. at the current delta cycle. The use of delta cycles emulates the parallel execution of processes. A process, running at the current delta cycle, can wake up other processes through an immediate event. Another variant is a signal update which will be executed during the next delta cycle. Again, these processes can trigger further ones at the subsequent delta cycle.

A detailed introduction into SystemC is given in the SystemC Language Reference Manual [IE+06] or can be found in [BDBK08].

1.3 Transaction Level Modeling in SystemC

With the publication of the OSCI TLM2 standard in 2008 [TLM2], SystemC took a major step forward to facilitate early system exploration, codesign of hardware and software, and platform-based design and verification. Moreover, a TLM modeling style improves the reusability of IP components between different IP providers. A standardized *Application Programming Interface* (API) allows the separation of computation inside components from the communication among components. The communication is modeled by channels where a transaction is released by calling interface methods of the channel. Unwanted communication details are hidden in the channels that are iteratively refined during modeling. Hence, system design starts with an abstract TLM description which enables a faster simulation as a comparable but more detailed RTL model. A predefined TLM interface is implemented by a high-level communication model initially describing only the exchange of messages or data between components. During development, the same interface can be implemented by a cycle-accurate *Bus Functional Model* (BFM) specifying the behavior of each pin later implemented in hardware. A BFM enables the designer to write test cases in terms of abstract method calls at TLM level. Then, the BFM translates the method call into particular input stimuli at the RTL site.

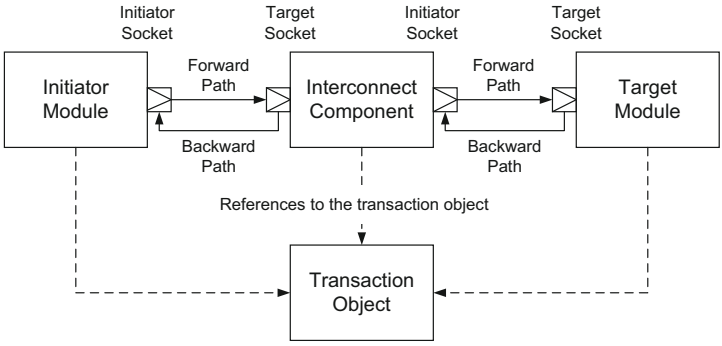

Figure 2.3: TLM notion in an example (taken from [TLM2])

The SystemC TLM2 standard distinguishes between coding styles and interfaces instead of defining abstraction levels for each particular use case. A use case can be for instance software development, software performance analysis, or hardware verification. The coding styles guide the designer in system modeling where the interfaces define low-level programming mechanisms. The TLM2 standard defines two coding styles: *loosely-timed*, and *approximately-timed* that are supported by particular blocking and non-blocking transport interfaces. Figure 2.3 documents an exemplary TLM design. A module can act in three different ways:

- *Initiator*. This module type creates new transactions and passes them to the channel by calling a predefined interface method.
- *Target*. A module of this type receives transactions and executes them according to the target module task.
- *Interconnect*. This component type forwards a transaction and possibly modifies it. So it acts as initiator and target at the same time.

The transportation path of a transaction that is going from an initiator to a target, is also called the *forward path*. The opposite direction is named the *backward path*. Over the backward path, the target informs the initiator about the transportation state. Either the modified transaction object is returned or a specific backward method is called explicitly. Two socket types encapsulate the connection between components. The *initiator socket* enables interface calls on the forward path by a *port* and on the backward path by an *export*. The *target socket* offers the same mechanism in case of a backward path. For a complete documentation of the TLM2 standard refer to [TLM2]. A comprehensive overview about the state-of-the-art of TLM design in SystemC can be also found in [Ghe06].

2 ESL VERIFICATION

Verification of functional design correctness has become the dominating cost and time factor in electronic system design, today. The correctness of a system has to be checked at each design step to identify errors as early as possible during system development. The later an error is detected, the more the costs for its correction increase. To cope with the rising verification complexity, a wide range of advanced techniques is used such as static analysis, debugging techniques, formal verification, assertion-based verification, constraint-random simulation, automatic test pattern generation, or model-based specification approaches. As already discussed on page 9, each abstraction level provides a defined degree of detail to fulfill a certain modeling or verification task, e.g. to measure the software performance or to explore different architectural choices. Starting with a system model, the model is refined step-by-step until the system is completed in the final software and hardware. An efficient abstraction mechanism for ESL design is provided by the introduction of SDLs together with a TLM design methodology. TLM facilitates the separation of computation and communication and allows to refine different design parts independently of each other (see Section 1.3 on page 16).

It is promising to extensively check higher level, less complex, descriptions before refining them. So, the detection and correction of an error is more efficient. A system design passes many refinement steps where in case of an error the design has to be fixed. Sometimes taken refinements are backtracked starting at a previous step, again. *Equivalence checking* ensures the equivalence between the current implementation and the specification. Does the implementation satisfy the specification, it will become the specification for the following design step. Writing the initial specification is a time-consuming and error-prone process. An imperfect specification increases the probability for *false positives*, i.e. pointing out non-bugs as bugs, or *false negatives*, i.e. failing to find real bugs. So, it is crucial to start with a correct and preferably complete specification which describes all important and desired aspects of the system.

The availability of an ESL design flow does not only allow the modeling of complex hardware. Moreover, the system model enables the designer to develop and to verify embedded software in parallel with the hardware. There, the hardware blocks can be exercised from the software programmers perspective. So, many functional errors can be found before the actual hardware is available. A system model has a further advantage. During development, parts of the system are replaced by their particular implementation (in hardware) whereas the rest becomes the verification environment. Thus, the hardware designer can be relieved from any additional writing of the verifica-

tion environment. Such a verification approach is applicable to find system-wide problems instead of verifying the correct function of a particular hardware block. It is also known as golden reference model verification where the system model defines the specification for the hardware components.

System correctness can be ensured by two different approaches: dynamic techniques, i.e. *simulation*, and static techniques, i.e. *formal verification*. In between, there is a third approach, that combines simulation and formal verification using *assertions*. It is called *semi-formal verification*. Moreover, there are further verification techniques, such as prototyping or emulation, but these techniques shall not be discussed in this book.

2.1 Simulation

Simulation is still the predominant verification technique in system and hardware design. It is easy to use, scales very well with arbitrary complex designs, and detects many functional errors very efficiently. As soon as an executable specification is available, it can be simulated easily. Besides the check of functional correctness, simulation is used to identify performance problems, inconsistencies, or specification holes. Apart from these advantages, the complexity of current designs prevents an exhaustive simulation to prove system correctness in a limited time [ARL00], [ITRS07]. Here, an increasing number of state variables let exponentially rise the number of input patterns, and thus the number of required simulation runs. So, the quality of simulation-based approaches highly depends on the quality of available input patterns to gain a satisfying coverage of the design functionality. Due to its incompleteness simulation is also named *validation*.

2.2 Formal Verification

To overcome the limitations of simulation, formal verification techniques were developed [Kro99]. Formal verification proves the correctness of a system using a mathematical model of the system behavior. This technique is exhaustive, and thus ensures correctness. Three different approaches are used in formal verification: *theorem proving* [WOLB92], *equivalence checking* [KPKG02], and *model checking*, also called *property checking* [CGP00].

In theorem proving the verified system (the implementation) and the specification are described in a higher-order logic. With the help of axioms and inference rules, a theorem prover supports the designer in reasoning that the implementation implies the specification. Since higher-order logics are generally undecidable, a fully automated proving is not possible. The required manual intervention has prevented a widespread application of theorem proving in the industrial field, so far.

Equivalence checking verifies the equivalence between implementation and specification. Normally, this technique verifies the correctness of (automatic) synthesis steps, e.g. by proving the equivalence between an RTL design and the synthesized gate level description. A common approach uses a miter circuit where two automata representations, one for the implementation and one for the specification, are combined. Then, the outputs are checked for equivalence while feeding in the same input data [Bra83]. The equivalence checking problem can be solved by transforming the miter circuit into a SAT instance. Today, equivalence checking enjoys a wide distribution in semiconductor industry especially due to its good automation and easy handling.

Model checking bases upon a finite state space model of the verified system, e.g. a labeled transition system or a finite state machine. The specification is formulated in terms of a set of properties in some temporal logic. Then, a model checker formally proves the validity of each (possibly unbounded) property on the model. *Bounded Model Checking* (BMC) limits the number of time frames a property is checked. Since a system usually responds after a finite time limit, BMC is quite efficient. It is successfully applied in the semiconductor industry, e.g. [BB+07]. A restriction of model checking is the limitation of a straightforward model translation (design abstraction) for large designs due to the so called *state space explosion* problem. Assuming that an example design consists of many thousand lines of code and for instance 5,000 32 bit integer variables. In that case, a Boolean reasoning technique has to handle 5,000 * 32 = 160,000 Boolean signals over many thousand program states. This is a potentially infeasible number to be currently handled by formal verification tools. Hence, classical model checking cannot verify arbitrary complex systems. New approaches try to use sophisticated abstraction techniques to handle the state explosion problem. *Satisfiability Modulo Theories* (SMT) solvers [NOT06] process word-level information directly. Another problem of model checking results from verifying only an abstracted model of the design. Hence, an erroneous abstraction process can produce false negatives as well as false positives. Moreover, a wrong specification could show that the system works as specified, but not as primarily intended. Finally, there are problematic structures, such as multipliers, that cannot be formally verified easily.

2.3 Semi-Formal Verification

Semi-formal verification combines simulation (see Section 2.1 on page 19) and formal verification (see Section 2.2 on page 19) and is also known as *Assertion-Based Verification* (ABV) [FKL03]. Assertions are temporal properties that concisely capture design intent. They are dynamically monitored during simulation. Two languages gain importance for applying ABV tech-

niques and allow the exact description of temporal-logic expressions: *SystemVerilog Assertions* (SVA) [IE+05a] and the *Property Specification Language* (PSL) [IE+05b]. To enable a fast and easy creation of an ABV test suite, the *Open Verification Library* (OVL) [OVL] captures typical design behavior in terms of assertion checkers.

ABV does not prove the specified property like in formal verification. For this reason, ABV can be applied to arbitrary complex designs. Additionally, assertions improve design observability, and thus facilitate the debugging process. In case of an error, an assertion directly points to the location the problem was recognized. This approach is also called white-box verification instead of the simulation oriented black-box approach. In simulation the response of a system is checked for correctness only at the output interfaces.

2.4 Verifying SystemC Models

SystemC only supports simulation natively while formal and semi-formal verification approaches are subject to current research. The following section summarizes important work in the particular fields.

2.4.1 Simulation in SystemC

The simulation-based validation is inherently supported by SystemC. Every standard C++ compiler, e.g. the GNU C++ compiler, is able to compile an arbitrary SystemC description into an executable program. Such a program can be directly simulated since the simulation kernel is an integral part of the SystemC library.

The processing of a SystemC simulation is particularly influenced by the style of coding. Several coding styles allow a different modeling of communication and synchronization between concurrent processes. SystemC FIFOs and semaphores introduce communication and synchronization points into the application. This style is formalized by *Communicating Sequential Processes* and *Kahn Process Networks*. It permits a completely *untimed* modeling style without the need for advancing the time during simulation. A coding style where each process yields control at a certain point in time is called *timed*. There, different timing levels can be distinguished such as loosely-timed, approximately timed, or cycle-accurate. A timed coding style is supported in SystemC by explicit synchronization statements.

Simulation uses directed or randomized tests to compare the expected results with the observed system behavior. A directed test validates only a very certain functionality by using an explicitly specified stimulus pattern. Usually, it is manually written by the designer which is a time-consuming and erroneous task. A major improvement in test bench automation was achieved

by the introduction of constraint-based random simulation [YPA06]. Based on a set of user-defined constraints, stimulus patterns are generated automatically. Hence, in a short time many more scenarios can be created and tested if compared to directed tests especially in case of corner cases. Constraint random simulation is supported in SystemC by the *SystemC Verification Library* (SCV) [SCV]. Moreover, this library provides other sophisticated verification features such as transaction recording and monitoring, or data introspection capabilities for arbitrary data types. Some work extents the SCV library. Große et al. [GED07] add bit operators and improve the integrated constraint solver to guarantee a uniform distribution of constraint solutions. Another work [GWSD08] introduces an approach that resolves contradictions between constraints automatically.

Based upon the SCV, various work propose verification frameworks or methodologies which allow for an easy, fast, and reusable simulation-centered functional verification of SystemC designs, e.g. [SMA04], [PC05]. These frameworks provide a unique verification infrastructure. They facilitate test bench creation and adaptation to a new design under verification, supply sophisticated coverage mechanisms, or integrate a constraint-random stimulus generator.

Functional code coverage techniques allow the designer to control simulation effort. Here, metrics provide a basis for decision in order to determine whether a SystemC design was simulated sufficiently. Various metrics measure code coverage using branch, statement, or condition coverage for instance. Traditional software metric tools for C++, such as gcov [GCOV], can be also applied to SystemC. The drawbacks of these tools are amongst others that coverage analysis also includes the SystemC kernel library routines. Furthermore, a C++ coverage tool does not know the SystemC semantics which requires a manual and tedious extraction of interesting coverage data. Große et al. [GPKD08] propose an approach that measures the quality of SystemC test benches in terms of a control flow metric. Therefore, an instrumented SystemC description collects coverage information during simulation. Subsequently, coverage information are analyzed whether the defined coverage goal has been already reached.

2.4.2 Semi-Formal Verification in SystemC

Since the SystemC standard does not define a native support for ABV, various approaches propose an integration of assertions into SystemC.

One of the first approaches was introduced by Ruf et al. [RHKR01]. It uses finite linear temporal logic to specify properties. These properties are translated into a special kind of finite-state machine, i.e. a monitor, in a preprocessing

phase. During simulation a monitor dynamically checks simulation behavior and reports each violation immediately.

The integration of SVA into SystemC is proposed by Habibi and Tahar [HT04]. An SVA expression is translated into an external SystemC module. This module acts as a monitor to all signals involved in the formulated assertion. Using the symbol table of a compiled SystemC design allows to connect the external monitor component with the design automatically. A so called design updater modifies the SystemC code in order to link design code and assertion monitor. During simulation, all SVA expressions are checked on-the-fly. A subsequent work [HT06] presents a model-driven design approach. Here, properties are modeled by extending UML sequence diagrams in order to check transaction properties of a SystemC TLM design. Therefore, the UML model of a PSL property is translated to an *Abstract State Machine* (ASM) description. Then, the ASM description is checked against the SystemC model which has to be available as an ASM model, as well. Both models are further translated to SystemC code where the property is compiled into a C# monitor.

Große and Drechsler [GD04a] present a method where a property (assertion) is translated into a synthesizable SystemC checker. The checker is embedded into the original SystemC description. Due to the synthesizable characteristics of checkers, they can be also evaluated after fabrication of the system.

Bombieri et al. [BFF05] evaluate the adoption of PSL to a SystemC-based TLM verification flow. They distinguish between two techniques, “properties re-use” and “properties refinement”. Properties re-use is utilized to ensure the functional equivalence of two different SoC descriptions. Properties refinement denotes the process of translating a PSL property, written for a more abstract design block, to the refined block. There, the PSL to SystemC checker translation is based upon the approach described in [DG+05].

Niemann and Haubelt [NH06] allow the concise formulation of assertions for TLM designs by using SVA expressions and interpret transactions as Boolean signals. To prevent a modification of the original source code, an aspect-oriented programming technique is used. Hence, transaction recording statements are weaved into the original SystemC design applying the AspectC++ compiler presented in [SLU05]. Then, the instrumented code is compiled and simulated. Transaction activity is written into a *Value Change Dump* (VCD) trace file. Subsequently, the VCD file is translated into a Verilog module which is simulated together with the formulated set of SVA assertions in a classical HDL simulator.

Kasuya and Tesfaye [KT07] introduce a native assertion mechanism for SystemC called NSCa. The designer creates assertions by using cycle-level

temporal primitives comparable with assertions written in SVA or PSL. NSCa assertions are written either as a separate file or they are embedded into the SystemC design. Moreover, the provided temporal primitives are applicable for higher levels of design abstraction, i.e. at the algorithmic or the TLM level using events and a simple queue model.

The largest problem for a widespread application of ABV techniques in SystemC is the missing standardization. The application of the introduced approaches

- is limited to a subset of SystemC, e.g. synthesizable descriptions [GD04a],
- introduces a new proprietary assertion language [KT07],
- integrates assertions only indirectly using additional tools [NH06], [RHKR01], [BFF05], [HT04], or
- does not completely support the underlying assertion language [HT06].

2.4.3 Formal Verification in SystemC

Formal verification gets a rising interest in SystemC design. Apart from the object-oriented nature of SystemC and its event-driven simulation semantics, the integration of software and hardware modules in the same system model makes the formal verification of SystemC designs a challenging task. Various approaches provide a formal simulation semantic for SystemC, e.g. [MR+01], [HT05], [Sal03]. However, a full formal semantics does not yet exist which complicates the development of formal techniques.

In [DG02] Drechsler and Große propose a reachability algorithm for sequential circuits described in SystemC. The algorithm bases upon *Binary Decision Diagrams* (BDD). Due to the limitations to synchronous sequential circuits, more abstract system descriptions are not supported by this approach. Based on the presented symbolic reachability algorithm, an approach for the formal verification of properties specified in *Linear Temporal Logic* (LTL) is given in [GD03].

Kröning and Sharygina [KS05] introduce a technique that automatically partitions a system model into a synchronous hardware part and an asynchronous software part. Partitioning is performed by a syntactical distinction of thread types (combinational vs. clocked threads). Labeled Kripke structures formalize the semantics of SystemC. Due to the partitioning step, the verified model contains fewer transitions, and thus is more efficiently verifiable.

Habibi and Tahar translate SystemC models into an intermediate representation using *Abstract State Machines Language* (AsmL) in order to support

model checking [HT06]. Using AsmL, the complexity of a SystemC design can be radically reduced which enables the verification of complex designs such as a PCI bus [OHT04]. The correct mapping between SystemC and AsmL is proven in [HT05].

Moy et al. enable the verification of SoCs described at transactional level. In [MMM05a] they introduce their toolbox LusSy that translates a SystemC design into a set of parallel automata that rely on a proprietary intermediate representation called HPIOM. HPIOM defines an executable formal semantics for TLM-based SystemC descriptions. Furthermore, it is connected to various formal tools such as the symbolic model checker LESAR. The translation rules are integrated into the GNU C++ compiler front end.

Herber et al. [HFG08] present an approach that defines the semantics of SystemC by a mapping from SystemC descriptions into Uppaal timed automata. These automata enables model checking in order to verify properties such as liveness, deadlock freedom or compliance with timing constraints.

The weakness of all formal verification approaches is their limitation in handling arbitrary complex SystemC designs. A missing formal semantics for the full language restricts the application of formal techniques in an industrial context. Mostly, the presented approaches support only a subset of SystemC features. Especially the use of object-oriented software concepts and the crucial pointer semantic cause the same problems model checking of C++ programs has. Although formal verification approaches promise to prove the correctness of system designs, particularly the complexity combined with a rising ratio of software prevent their widespread and straightforward usage. So, classical simulation still remains the means of choice. Nevertheless, new techniques, such as ABV, bridge the gap between simulation and formal verification, and try to combine the benefits of both worlds.

3 OUR DEBUGGING APPROACH

At first, this section defines common terms used in the context of debugging in this book. Next, the general debugging process is sketched. Afterwards, our debugging approach, consisting of various debugging techniques, is described. Finally, an example which is used throughout the book is introduced.

3.1 Terms

Debugging is the process of detecting and fixing a defect that has caused an observed failure. To clarify the different terms used in this context, i.e.

defect and *failure*, we want to define them in the same way as Zeller has done in [Zel05]:

- A *defect*, also known as *bug* or *fault*, is a faulty location in the program code that can cause an infection.
- An *infection* is an error in the program state that can cause a failure if this state is propagated.
- A *failure* is an externally visible situation in the program behavior that differs from the expected behavior.

It should be noted that a failure can be produced by a combination of defects where only the combination leads to an infection. Simultaneously, the same defect can produce many different failures.

3.2 General Debug Process

Debugging in the hardware as well as in the software domain can be mapped to the following general procedure:

1. Exactly describe the observed problem. Sometime, you can already catch the problem cause while revising the erroneous situation in detail.
2. Ask the question whether the problem could be a software failure?
   a. If yes

      reproduce the failure by simplifying and automating the failure-causing test case,

      locate the failure-causing bug, e.g. constitute a hypothesis and test it, observe the program state, isolate the correlation between cause and effect, and fix the bug.
   b. If no,

      Find the problem cause and fix it. The defect could be for instance a hardware problem, a bug in the used compiler, or a simple misunderstanding in handling the software.
3. Verify the effectiveness of the bug fix.
   a. Could the entire problem be solved?
   b. Are there new, formerly unknown problems?

3.3 Hierarchy of Debugging Techniques

The main objective of the book is the development of a *systematic debugging approach* that enables a methodic search for errors while verifying ESL designs. So, errors can be detected and located efficiently at the different development stages of the system model. As a result, the approach contributes to an improvement of the overall verification productivity. It starts with first testable modules. Here, static analysis identifies coding flaws and potential failure causes. At the end, a controlled dynamic analysis automates debugging of the completed system model. The application of one of the debugging techniques does not depend on the current used abstraction level (see Figure 1.4). So, static analysis can be utilized on untimed as well as cycle-accurate model descriptions. Rather, the classification of the techniques is made by the number of needed simulation runs, the reached realization level, and the achieved coding level (see Figure 1.4). The debugging techniques form a reasoning hierarchy as proposed by Zeller [Zel05]. Zeller introduces a hierarchy of program analysis techniques to provide a classification of debugging techniques especially used for software programs. This hierarchy shall help the programmer to systemize the overall debugging process:

- *Deduction*. Deduction denotes the reasoning from the statically analyzed program code to the concrete program run. Generally, it means the reasoning from the general to the particular. This reasoning technique bases upon an abstraction of the program that is used to deduce particular properties.
- *Observation*. Observation is the first dynamic analysis technique. There, a particular program run is used to explore arbitrary program aspects. In contrast to deduction, this technique observes the results of a single program run and tries to find approximations based on that run.
- *Induction*. In general, induction tries to infer from particular observations to the general program behavior. Actually this means, that many program runs are merged to a general abstraction which holds for all runs.
- *Experimentation*. Experiments are used to search for the cause of an observed failure systematically. A series of experiments is created that tests a hypothesis with the goal to reject or confirm it. As a result, a precise diagnosis shall be isolated using a number of controlled program runs.

In this book, the reasoning hierarchy shall be adapted to the specific requirements and conditions of debugging ESL designs in terms of SystemC model descriptions:

- *Deduction*. As soon as the designer has written syntactical correct source code, hypotheses can be deduced from the code without the need for simulating the model. A static analyzer for SystemC, which is based on a generic static analysis framework, performs different analyses, This concerns an analysis of coding standards and the check for functional errors.
- *Observation*. The observation of a particular simulation run of the developed system model is supported by a high-level debug flow. Various sophisticated debugging and exploration features support the designer in exploring the actual simulation state. A SystemC debugger, debug patterns, and visualization features allow a fast error detection, location, and correction.
- *Induction*. Multiple simulation runs generate a set of simulation traces. The traces are used to deduce from a number of concrete runs general abstractions by means of design properties. Properties, that are generated on the golden reference model, can be later used to check the functional correctness of more refined design models, e.g. the hardware part at RTL.
- *Experimentation*. Using a set of controlled simulation runs allows to isolate failure-inducing causes in a system model. The delta debugging algorithm of Zeller and Hildebrandt [ZH02] is adapted to SystemC to automate debugging of design descriptions.

Figure 2.4 illustrates the classification of the introduced debugging techniques in an ESL design flow (see Figure 2.1). There, a technique shall be used as soon as the preconditions for its usage are given (see Figure 1.4). Each technique is implemented by a prototypical tool for SystemC to demonstrate its effectiveness and efficiency. Generally, the techniques aim at a fast and easy location of errors in the system model.

3.4 SIMD Data Transfer Example

An example, that is used throughout the book, shall demonstrate the particularities and strengths of each introduced debugging technique. The example models the simplified data transfer between the cache of a *Processor Unit* and a global *Shared Memory* of an *Single Instruction Multiple Data* (SIMD) processor design. Figure 2.5 shows the general architecture of the

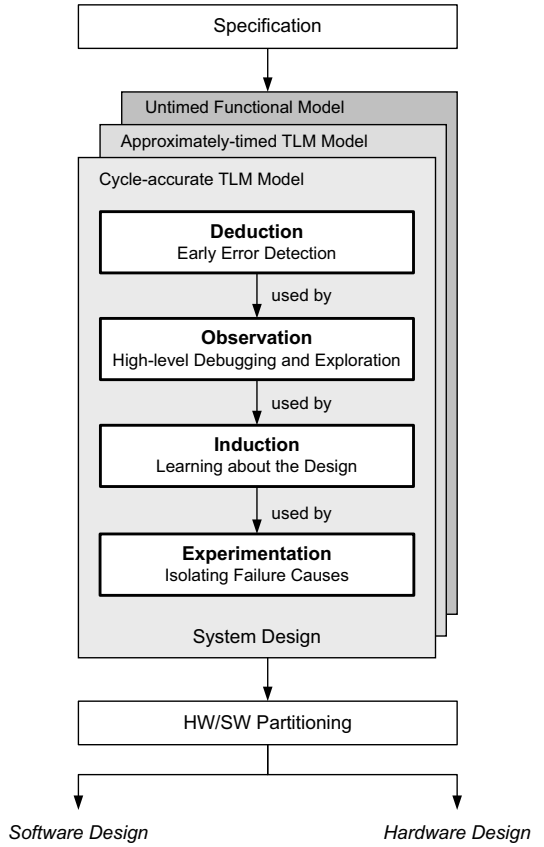

Figure 2.4: Reasoning hierarchy in a simplified ESL design flow

SIMD data transfer example. Here, only a single processor core of the entire processor design is considered. The synchronization mechanisms between the different cores are omitted for simplification reasons. The data transfer is represented by generic payload transactions conforming to the SystemC TLM2 standard [TLM2]. The memory access is handled by a *Load/Store Controller* attached to the particular processor unit. Here, an integrated arbiter controls and handles the read and write transfers from/to the shared memory. Data are either read or written, i.e. there is a mutual exclusion between read and write accesses due to a specified single port cache structure. An internal buffer in the load/store controller decouples memory and cache. The external *Monitor* component supports debugging tasks. It records the transaction activities and writes them into a VCD dump file.

Figure 2.6 depicts an example read/write data transfer between the processor cache and the shared memory. Each data transfer starts with a read

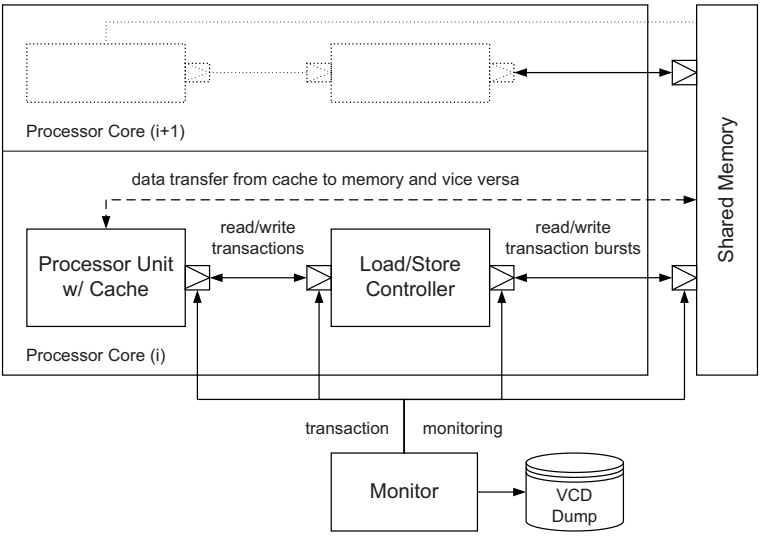

Figure 2.5: General architecture of the SIMD data transfer example

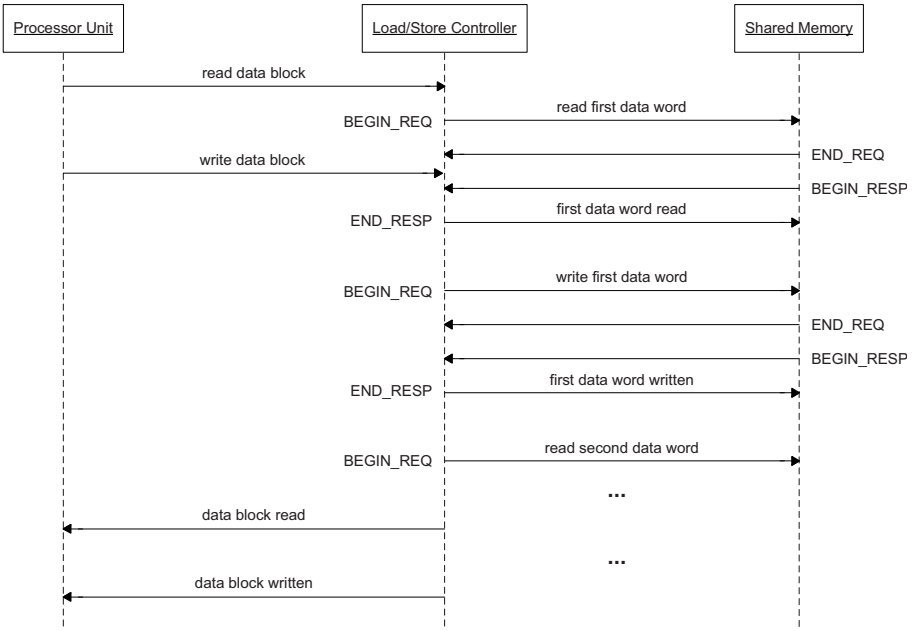

Figure 2.6: Example of a read/write data transfer sequence

transaction using the TLM2 blocking transport interface. It initiates the reading of a data block from a specified (random) address in the memory into the cache. Each block consists of a multiple of an integer word (4 bytes) for demonstration purposes. The load/store controller takes the read transaction and generates a number of burst transactions. Each burst transmits a single data word out of the data block from the memory. This behavior models the limited band width of the memory bus. The band width prevents a transfer of the data block as a whole. Every burst transaction is modeled by the TLM2 base protocol in the approximately-timed coding style using the four standard phases, i.e. BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP (see Figure 2.6). Hence, a "real" data transfer, using handshake signaling, can be described close to reality. In the real-world design, the cached data are subsequently processed by a loaded SIMD program. As soon as processed data are available, they are written back into memory. For the sake of simplicity, the test bench initiates the write transfer after a random time and writes the cached data to a specified (random) memory address. Write transfers are equally handled like read transfers. Due to the specified single port cache, it is not possible to handle bursts in parallel. So, data transfers take place in a mutual exclusive fashion which ensures fairness. Thus the overall processing time is optimized.

http://www.springer.com/978-90-481-9254-0

Debugging at the Electronic System Level
Rogin, F.; Drechsler, R.
2010, XIX, 199 p., Hardcover
ISBN: 978-90-481-9254-0